

Internet Engineering Task Force  
Internet-Draft  
Obsoletes: 793, 879, 6093, 6528, 6691  
(if approved)  
Updates: 1122 (if approved)  
Intended status: Standards Track  
Expires: August 10, 2015

W. Eddy, Ed.  
MTI Systems  
February 6, 2015

Transmission Control Protocol Specification  
draft-eddy-rfc793bis-05

Abstract

This document specifies the Internet's Transmission Control Protocol (TCP). TCP is an important transport layer protocol in the Internet stack, and has continuously evolved over decades of use and growth of the Internet. Over this time, a number of changes have been made to TCP as it was specified in RFC 793, though these have only been documented in a piecemeal fashion. This document collects and brings those changes together with the protocol specification from RFC 793. This document obsoletes RFC 793 and several other RFCs (TODO: list all actual RFCs when finished).

RFC EDITOR NOTE: If approved for publication as an RFC, this should be marked additionally as "STD: 7" and replace RFC 793 in that role.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 10, 2015.

#### Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

#### Table of Contents

1.	Purpose and Scope . . . . .	3
2.	Introduction . . . . .	4
3.	Functional Specification . . . . .	4
3.1.	Header Format . . . . .	4
3.2.	Terminology . . . . .	9
3.3.	Sequence Numbers . . . . .	13
3.4.	Establishing a connection . . . . .	20
3.5.	Closing a Connection . . . . .	27
3.6.	Precedence and Security . . . . .	29
3.7.	Segmentation . . . . .	30
3.7.1.	Maximum Segment Size Option . . . . .	31
3.7.2.	Path MTU Discovery . . . . .	32
3.7.3.	Interfaces with Variable MSS Values . . . . .	32
3.7.4.	IPv6 Jumbograms . . . . .	32
3.8.	Data Communication . . . . .	32
3.9.	Interfaces . . . . .	36
3.9.1.	User/TCP Interface . . . . .	37

3.9.2. TCP/Lower-Level Interface . . . . .	43
3.10. Event Processing . . . . .	44
3.11. Glossary . . . . .	67
4. Changes from RFC 793 . . . . .	72
5. IANA Considerations . . . . .	75
6. Security and Privacy Considerations . . . . .	75
7. Acknowledgements . . . . .	76
8. References . . . . .	76
8.1. Normative References . . . . .	76
8.2. Informative References . . . . .	76
Appendix A. TCP Requirement Summary . . . . .	77
Author's Address . . . . .	80

## 1. Purpose and Scope

In 1981, RFC 793 [2] was released, documenting the Transmission Control Protocol (TCP), and replacing earlier specifications for TCP that had been published in the past.

Since then, TCP has been implemented many times, and has been used as a transport protocol for numerous applications on the Internet.

For several decades, RFC 793 plus a number of other documents have combined to serve as the specification for TCP [10]. Over time, a number of errata have been identified on RFC 793, as well as deficiencies in security, performance, and other aspects. A number of enhancements has grown and been documented separately. These were never accumulated together into an update to the base specification.

The purpose of this document is to bring together all of the IETF Standards Track changes that have been made to the basic TCP functional specification and unify them into an update of the RFC 793 protocol specification. Some companion documents are referenced for important algorithms that TCP uses (e.g. for congestion control), but have not been attempted to include in this document. This is a conscious choice, as this base specification can be used with multiple additional algorithms that are developed and incorporated separately, but all TCP implementations need to implement this specification as a common basis in order to interoperate. As some additional TCP features have become quite complicated themselves (e.g. advanced loss recovery and congestion control), future companion documents may attempt to similarly bring these together.

In addition to the protocol specification that describes the TCP segment format, generation, and processing rules that are to be implemented in code, RFC 793 and other updates also contain informative and descriptive text for human readers to understand aspects of the protocol design and operation. This document does not

attempt to alter or update this informative text, and is focused only on updating the normative protocol specification. We preserve references to the documentation containing the important explanations and rationale, where appropriate.

This document is intended to be useful both in checking existing TCP implementations for conformance, as well as in writing new implementations.

## 2. Introduction

RFC 793 contains a discussion of the TCP design goals and provides examples of its operation, including examples of connection establishment, closing connections, and retransmitting packets to repair losses.

This document describes the basic functionality expected in modern implementations of TCP, and replaces the protocol specification in RFC 793. It does not replicate or attempt to update the examples and other discussion in RFC 793. Other documents are referenced to provide explanation of the theory of operation, rationale, and detailed discussion of design decisions. This document only focuses on the normative behavior of the protocol.

TEMPORARY EDITOR'S NOTE: This is an early revision in the process of updating RFC 793. Many planned changes are not yet incorporated.

\*\*\*Please do not use this revision as a basis for any work or reference.\*\*\*

A list of changes from RFC 793 is contained in Section 4.

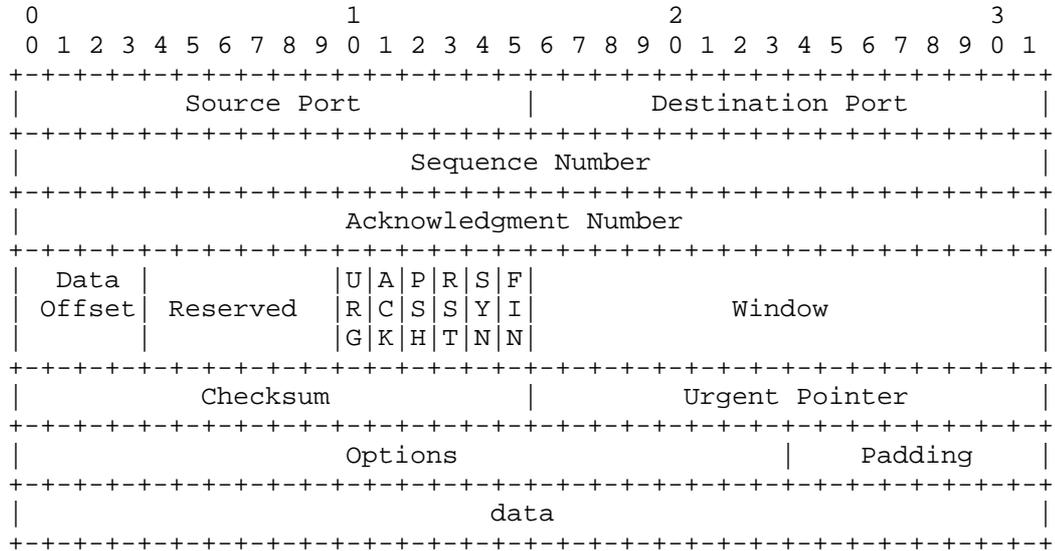
TEMPORARY EDITOR'S NOTE: the current revision of this document does not yet collect all of the changes that will be in the final version. The set of content changes planned for future revisions is kept in Section 4.

## 3. Functional Specification

### 3.1. Header Format

TCP segments are sent as internet datagrams. The Internet Protocol header carries several information fields, including the source and destination host addresses [2]. A TCP header follows the internet header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP.

TCP Header Format



TCP Header Format

Note that one tick mark represents one bit position.

Figure 1

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.

Sequence Number: 32 bits

The sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.

Acknowledgment Number: 32 bits

If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.

Data Offset: 4 bits

The number of 32 bit words in the TCP Header. This indicates where the data begins. The TCP header (even one including options) is an integral number of 32 bits long.

Reserved: 6 bits

Reserved for future use. Must be zero.

Control Bits: 6 bits (from left to right):

- URG: Urgent Pointer field significant
- ACK: Acknowledgment field significant
- PSH: Push Function
- RST: Reset the connection
- SYN: Synchronize sequence numbers
- FIN: No more data from sender

Window: 16 bits

The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept.

Checksum: 16 bits

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16 bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.

The checksum also covers a 96 bit pseudo header conceptually prefixed to the TCP header. This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the TCP protection against misrouted segments. This information is carried in the Internet Protocol and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.

```

+-----+-----+-----+-----+
|           Source Address           |
+-----+-----+-----+-----+
|           Destination Address      |
+-----+-----+-----+-----+
| zero | PTCL |   TCP Length   |
+-----+-----+-----+-----+

```

The TCP Length is the TCP header length plus the data length in octets (this is not an explicitly transmitted quantity, but is computed), and it does not count the 12 octets of the pseudo header.

Urgent Pointer: 16 bits

This field communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only be interpreted in segments with the URG control bit set.

Options: variable

Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum. An option may begin on any octet boundary. There are two cases for the format of an option:

Case 1: A single octet of option-kind.

Case 2: An octet of option-kind, an octet of option-length, and the actual option-data octets.

The option-length counts the two octets of option-kind and option-length as well as the option-data octets.

Note that the list of options may be shorter than the data offset field might imply. The content of the header beyond the End-of-Option option must be header padding (i.e., zero).

Currently defined options include (kind indicated in octal):

Kind	Length	Meaning
----	-----	-----
0	-	End of option list.
1	-	No-Operation.
2	4	Maximum Segment Size.

A TCP MUST be able to receive a TCP option in any segment. A TCP MUST ignore without error any TCP option it does not implement, assuming that the option has a length field (all TCP options except End of option list and No-Operation have length fields). TCP MUST be prepared to handle an illegal option length (e.g., zero) without crashing; a suggested procedure is to reset the connection and log the reason.

#### Specific Option Definitions

##### End of Option List

```
+-----+
|00000000|
+-----+
Kind=0
```

This option code indicates the end of the option list. This might not coincide with the end of the TCP header according to the Data Offset field. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end of the TCP header.

##### No-Operation

```
+-----+
|00000001|
+-----+
Kind=1
```

This option code may be used between options, for example, to align the beginning of a subsequent option on a word boundary. There is no guarantee that senders will use this option, so receivers must be prepared to process options even if they do not begin on a word boundary.

##### Maximum Segment Size (MSS)

```
+-----+-----+-----+-----+
|00000010|00000100|  max seg size  |
+-----+-----+-----+-----+
Kind=2   Length=4
```

##### Maximum Segment Size Option Data: 16 bits

If this option is present, then it communicates the maximum receive segment size at the TCP which sends this segment. This

field may be sent in the initial connection request (i.e., in segments with the SYN control bit set) and must not be sent in other segments. If this option is not used, any segment size is allowed.

Padding: variable

The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.

### 3.2. Terminology

Before we can discuss very much about the operation of the TCP we need to introduce some detailed terminology. The maintenance of a TCP connection requires the remembering of several variables. We conceive of these variables being stored in a connection record called a Transmission Control Block or TCB. Among the variables stored in the TCB are the local and remote socket numbers, the security and precedence of the connection, pointers to the user's send and receive buffers, pointers to the retransmit queue and to the current segment. In addition several variables relating to the send and receive sequence numbers are stored in the TCB.

#### Send Sequence Variables

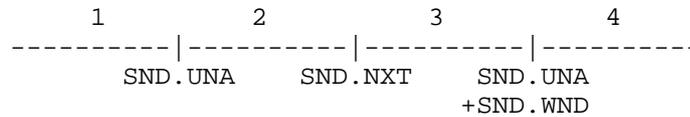
SND.UNA - send unacknowledged  
SND.NXT - send next  
SND.WND - send window  
SND.UP - send urgent pointer  
SND.WL1 - segment sequence number used for last window update  
SND.WL2 - segment acknowledgment number used for last window update  
ISS - initial send sequence number

#### Receive Sequence Variables

RCV.NXT - receive next  
RCV.WND - receive window  
RCV.UP - receive urgent pointer  
IRS - initial receive sequence number

The following diagrams may help to relate some of these variables to the sequence space.

## Send Sequence Space



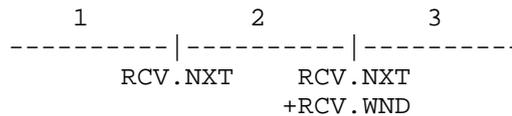
- 1 - old sequence numbers which have been acknowledged
- 2 - sequence numbers of unacknowledged data
- 3 - sequence numbers allowed for new data transmission
- 4 - future sequence numbers which are not yet allowed

## Send Sequence Space

Figure 2

The send window is the portion of the sequence space labeled 3 in Figure 2.

## Receive Sequence Space



- 1 - old sequence numbers which have been acknowledged
- 2 - sequence numbers allowed for new reception
- 3 - future sequence numbers which are not yet allowed

## Receive Sequence Space

Figure 3

The receive window is the portion of the sequence space labeled 2 in Figure 3.

There are also some variables used frequently in the discussion that take their values from the fields of the current segment.

## Current Segment Variables

- SEG.SEQ - segment sequence number
- SEG.ACK - segment acknowledgment number
- SEG.LEN - segment length
- SEG.WND - segment window
- SEG.UP - segment urgent pointer
- SEG.PRC - segment precedence value

A connection progresses through a series of states during its lifetime. The states are: LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT, and the fictional state CLOSED. CLOSED is fictional because it represents the state when there is no TCB, and therefore, no connection. Briefly the meanings of the states are:

LISTEN - represents waiting for a connection request from any remote TCP and port.

SYN-SENT - represents waiting for a matching connection request after having sent a connection request.

SYN-RECEIVED - represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

ESTABLISHED - represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

FIN-WAIT-1 - represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

FIN-WAIT-2 - represents waiting for a connection termination request from the remote TCP.

CLOSE-WAIT - represents waiting for a connection termination request from the local user.

CLOSING - represents waiting for a connection termination request acknowledgment from the remote TCP.

LAST-ACK - represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (this termination request sent to the remote TCP already included an acknowledgment of the termination request sent from the remote TCP).

TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

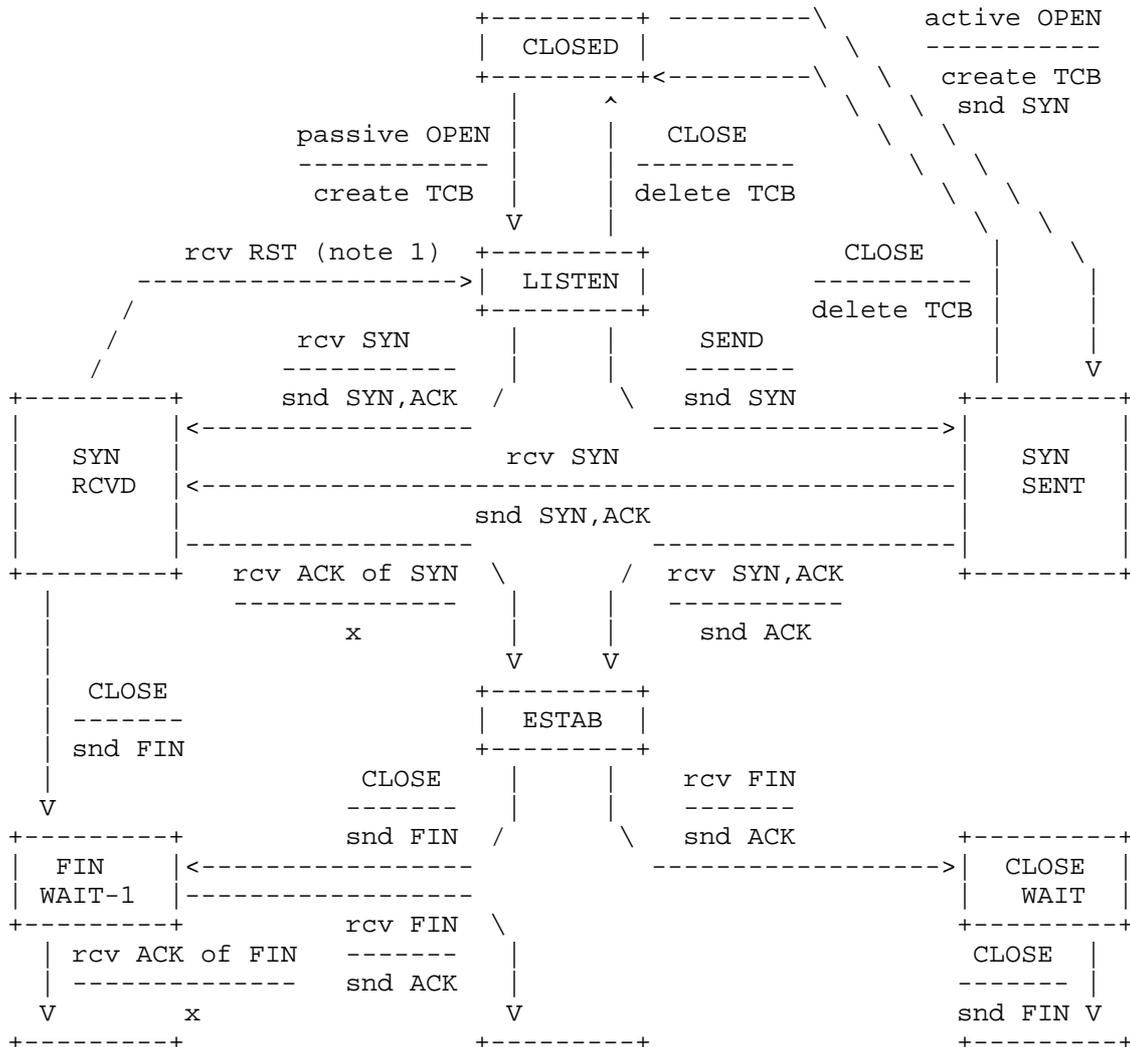
CLOSED - represents no connection state at all.

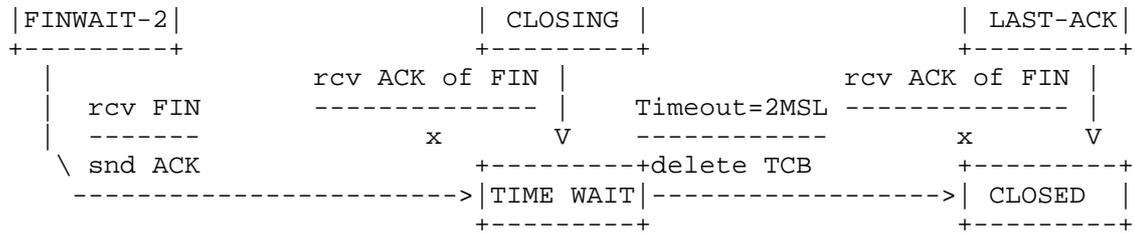
A TCP connection progresses from one state to another in response to events. The events are the user calls, OPEN, SEND, RECEIVE, CLOSE,

ABORT, and STATUS; the incoming segments, particularly those containing the SYN, ACK, RST and FIN flags; and timeouts.

The state diagram in Figure 4 illustrates only state changes, together with the causing events and resulting actions, but addresses neither error conditions nor actions which are not connected with state changes. In a later section, more detail is offered with respect to the reaction of the TCP to events.

NOTA BENE: this diagram is only a summary and must not be taken as the total specification.





note 1: The transition from SYN-RCVD to LISTEN on receiving a RST is conditional on having reached SYN-RCVD after a passive open.

note 2: An unshown transition exists from FIN-WAIT-1 to TIME-WAIT if a FIN is received and the local FIN is also acknowledged.

TCP Connection State Diagram

Figure 4

### 3.3. Sequence Numbers

A fundamental notion in the design is that every octet of data sent over a TCP connection has a sequence number. Since every octet is sequenced, each of them can be acknowledged. The acknowledgment mechanism employed is cumulative so that an acknowledgment of sequence number X indicates that all octets up to but not including X have been received. This mechanism allows for straight-forward duplicate detection in the presence of retransmission. Numbering of octets within a segment is that the first data octet immediately following the header is the lowest numbered, and the following octets are numbered consecutively.

It is essential to remember that the actual sequence number space is finite, though very large. This space ranges from 0 to  $2^{32} - 1$ . Since the space is finite, all arithmetic dealing with sequence numbers must be performed modulo  $2^{32}$ . This unsigned arithmetic preserves the relationship of sequence numbers as they cycle from  $2^{32} - 1$  to 0 again. There are some subtleties to computer modulo arithmetic, so great care should be taken in programming the comparison of such values. The symbol " $=<$ " means "less than or equal" (modulo  $2^{32}$ ).

The typical kinds of sequence number comparisons which the TCP must perform include:

- (a) Determining that an acknowledgment refers to some sequence number sent but not yet acknowledged.

(b) Determining that all sequence numbers occupied by a segment have been acknowledged (e.g., to remove the segment from a retransmission queue).

(c) Determining that an incoming segment contains sequence numbers which are expected (i.e., that the segment "overlaps" the receive window).

In response to sending data the TCP will receive acknowledgments. The following comparisons are needed to process the acknowledgments.

SND.UNA = oldest unacknowledged sequence number

SND.NXT = next sequence number to be sent

SEG.ACK = acknowledgment from the receiving TCP (next sequence number expected by the receiving TCP)

SEG.SEQ = first sequence number of a segment

SEG.LEN = the number of octets occupied by the data in the segment (counting SYN and FIN)

SEG.SEQ+SEG.LEN-1 = last sequence number of a segment

A new acknowledgment (called an "acceptable ack"), is one for which the inequality below holds:

$$\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$$

A segment on the retransmission queue is fully acknowledged if the sum of its sequence number and length is less or equal than the acknowledgment value in the incoming segment.

When data is received the following comparisons are needed:

RCV.NXT = next sequence number expected on an incoming segments, and is the left or lower edge of the receive window

RCV.NXT+RCV.WND-1 = last sequence number expected on an incoming segment, and is the right or upper edge of the receive window

SEG.SEQ = first sequence number occupied by the incoming segment

SEG.SEQ+SEG.LEN-1 = last sequence number occupied by the incoming segment

A segment is judged to occupy a portion of valid receive sequence space if

$$\text{RCV.NXT} \leq \text{SEG.SEQ} < \text{RCV.NXT} + \text{RCV.WND}$$

or

$$\text{RCV.NXT} \leq \text{SEG.SEQ} + \text{SEG.LEN} - 1 < \text{RCV.NXT} + \text{RCV.WND}$$

The first part of this test checks to see if the beginning of the segment falls in the window, the second part of the test checks to see if the end of the segment falls in the window; if the segment passes either part of the test it contains data in the window.

Actually, it is a little more complicated than this. Due to zero windows and zero length segments, we have four cases for the acceptability of an incoming segment:

Segment Length	Receive Window	Test
0	0	$\text{SEG.SEQ} = \text{RCV.NXT}$
0	>0	$\text{RCV.NXT} \leq \text{SEG.SEQ} < \text{RCV.NXT} + \text{RCV.WND}$
>0	0	not acceptable
>0	>0	$\text{RCV.NXT} \leq \text{SEG.SEQ} < \text{RCV.NXT} + \text{RCV.WND}$ or $\text{RCV.NXT} \leq \text{SEG.SEQ} + \text{SEG.LEN} - 1 < \text{RCV.NXT} + \text{RCV.WND}$

Note that when the receive window is zero no segments should be acceptable except ACK segments. Thus, it is possible for a TCP to maintain a zero receive window while transmitting data and receiving ACKs. However, even when the receive window is zero, a TCP must process the RST and URG fields of all incoming segments.

We have taken advantage of the numbering scheme to protect certain control information as well. This is achieved by implicitly including some control flags in the sequence space so they can be retransmitted and acknowledged without confusion (i.e., one and only one copy of the control will be acted upon). Control information is not physically carried in the segment data space. Consequently, we must adopt rules for implicitly assigning sequence numbers to control. The SYN and FIN are the only controls requiring this protection, and these controls are used only at connection opening and closing. For sequence number purposes, the SYN is considered to occur before the first actual data octet of the segment in which it

occurs, while the FIN is considered to occur after the last actual data octet in a segment in which it occurs. The segment length (SEG.LEN) includes both data and sequence space occupying controls. When a SYN is present then SEG.SEQ is the sequence number of the SYN.

#### Initial Sequence Number Selection

The protocol places no restriction on a particular connection being used over and over again. A connection is defined by a pair of sockets. New instances of a connection will be referred to as incarnations of the connection. The problem that arises from this is -- "how does the TCP identify duplicate segments from previous incarnations of the connection?" This problem becomes apparent if the connection is being opened and closed in quick succession, or if the connection breaks with loss of memory and is then reestablished.

To avoid confusion we must prevent segments from one incarnation of a connection from being used while the same sequence numbers may still be present in the network from an earlier incarnation. We want to assure this, even if a TCP crashes and loses all knowledge of the sequence numbers it has been using. When new connections are created, an initial sequence number (ISN) generator is employed which selects a new 32 bit ISN. There are security issues that result if an off-path attacker is able to predict or guess ISN values.

The recommended ISN generator is based on the combination of a (possibly fictitious) 32 bit clock whose low order bit is incremented roughly every 4 microseconds, and a pseudorandom hash function (PRF). The clock component is intended to insure that with a Maximum Segment Lifetime (MSL), generated ISNs will be unique, since it cycles approximately every 4.55 hours, which is much longer than the MSL.

TCP SHOULD generate its Initial Sequence Numbers with the expression:

$$\text{ISN} = M + F(\text{localip}, \text{localport}, \text{remoteip}, \text{remoteport}, \text{secretkey})$$

where M is the 4 microsecond timer, and F() is a pseudorandom function (PRF) of the connection's identifying parameters ("localip, localport, remoteip, remoteport") and a secret key ("secretkey"). F() MUST NOT be computable from the outside, or an attacker could still guess at sequence numbers from the ISN used for some other connection. The PRF could be implemented as a cryptographic hash of the concatenation of the TCP connection parameters and some secret data. For discussion of the selection of a specific hash algorithm and management of the secret key data, please see Section 3 of [8].

For each connection there is a send sequence number and a receive sequence number. The initial send sequence number (ISS) is chosen by

the data sending TCP, and the initial receive sequence number (IRS) is learned during the connection establishing procedure.

For a connection to be established or initialized, the two TCPs must synchronize on each other's initial sequence numbers. This is done in an exchange of connection establishing segments carrying a control bit called "SYN" (for synchronize) and the initial sequence numbers. As a shorthand, segments carrying the SYN bit are also called "SYNs". Hence, the solution requires a suitable mechanism for picking an initial sequence number and a slightly involved handshake to exchange the ISN's.

The synchronization requires each side to send it's own initial sequence number and to receive a confirmation of it in acknowledgment from the other side. Each side must also receive the other side's initial sequence number and send a confirming acknowledgment.

- 1) A --> B SYN my sequence number is X
- 2) A <-- B ACK your sequence number is X
- 3) A <-- B SYN my sequence number is Y
- 4) A --> B ACK your sequence number is Y

Because steps 2 and 3 can be combined in a single message this is called the three way (or three message) handshake.

A three way handshake is necessary because sequence numbers are not tied to a global clock in the network, and TCPs may have different mechanisms for picking the ISN's. The receiver of the first SYN has no way of knowing whether the segment was an old delayed one or not, unless it remembers the last sequence number used on the connection (which is not always possible), and so it must ask the sender to verify this SYN. The three way handshake and the advantages of a clock-driven scheme are discussed in [3].

#### Knowing When to Keep Quiet

To be sure that a TCP does not create a segment that carries a sequence number which may be duplicated by an old segment remaining in the network, the TCP must keep quiet for a maximum segment lifetime (MSL) before assigning any sequence numbers upon starting up or recovering from a crash in which memory of sequence numbers in use was lost. For this specification the MSL is taken to be 2 minutes. This is an engineering choice, and may be changed if experience indicates it is desirable to do so. Note that if a TCP is reinitialized in some sense, yet retains its memory of sequence numbers in use, then it need not wait at all; it must only be sure to use sequence numbers larger than those recently used.

### The TCP Quiet Time Concept

This specification provides that hosts which "crash" without retaining any knowledge of the last sequence numbers transmitted on each active (i.e., not closed) connection shall delay emitting any TCP segments for at least the agreed Maximum Segment Lifetime (MSL) in the internet system of which the host is a part. In the paragraphs below, an explanation for this specification is given. TCP implementors may violate the "quiet time" restriction, but only at the risk of causing some old data to be accepted as new or new data rejected as old duplicated by some receivers in the internet system.

TCPs consume sequence number space each time a segment is formed and entered into the network output queue at a source host. The duplicate detection and sequencing algorithm in the TCP protocol relies on the unique binding of segment data to sequence space to the extent that sequence numbers will not cycle through all  $2^{32}$  values before the segment data bound to those sequence numbers has been delivered and acknowledged by the receiver and all duplicate copies of the segments have "drained" from the internet. Without such an assumption, two distinct TCP segments could conceivably be assigned the same or overlapping sequence numbers, causing confusion at the receiver as to which data is new and which is old. Remember that each segment is bound to as many consecutive sequence numbers as there are octets of data and SYN or FIN flags in the segment.

Under normal conditions, TCPs keep track of the next sequence number to emit and the oldest awaiting acknowledgment so as to avoid mistakenly using a sequence number over before its first use has been acknowledged. This alone does not guarantee that old duplicate data is drained from the net, so the sequence space has been made very large to reduce the probability that a wandering duplicate will cause trouble upon arrival. At 2 megabits/sec. it takes 4.5 hours to use up  $2^{32}$  octets of sequence space. Since the maximum segment lifetime in the net is not likely to exceed a few tens of seconds, this is deemed ample protection for foreseeable nets, even if data rates escalate to 10's of megabits/sec. At 100 megabits/sec, the cycle time is 5.4 minutes which may be a little short, but still within reason.

The basic duplicate detection and sequencing algorithm in TCP can be defeated, however, if a source TCP does not have any memory of the sequence numbers it last used on a given connection. For example, if the TCP were to start all connections with sequence number 0, then upon crashing and restarting, a TCP might re-form an earlier connection (possibly after half-open connection resolution) and emit packets with sequence numbers identical to or overlapping with

packets still in the network which were emitted on an earlier incarnation of the same connection. In the absence of knowledge about the sequence numbers used on a particular connection, the TCP specification recommends that the source delay for MSL seconds before emitting segments on the connection, to allow time for segments from the earlier connection incarnation to drain from the system.

Even hosts which can remember the time of day and used it to select initial sequence number values are not immune from this problem (i.e., even if time of day is used to select an initial sequence number for each new connection incarnation).

Suppose, for example, that a connection is opened starting with sequence number  $S$ . Suppose that this connection is not used much and that eventually the initial sequence number function ( $ISN(t)$ ) takes on a value equal to the sequence number, say  $S_1$ , of the last segment sent by this TCP on a particular connection. Now suppose, at this instant, the host crashes, recovers, and establishes a new incarnation of the connection. The initial sequence number chosen is  $S_1 = ISN(t)$  -- last used sequence number on old incarnation of connection! If the recovery occurs quickly enough, any old duplicates in the net bearing sequence numbers in the neighborhood of  $S_1$  may arrive and be treated as new packets by the receiver of the new incarnation of the connection.

The problem is that the recovering host may not know for how long it crashed nor does it know whether there are still old duplicates in the system from earlier connection incarnations.

One way to deal with this problem is to deliberately delay emitting segments for one MSL after recovery from a crash- this is the "quiet time" specification. Hosts which prefer to avoid waiting are willing to risk possible confusion of old and new packets at a given destination may choose not to wait for the "quite time". Implementors may provide TCP users with the ability to select on a connection by connection basis whether to wait after a crash, or may informally implement the "quite time" for all connections. Obviously, even where a user selects to "wait," this is not necessary after the host has been "up" for at least MSL seconds.

To summarize: every segment emitted occupies one or more sequence numbers in the sequence space, the numbers occupied by a segment are "busy" or "in use" until MSL seconds have passed, upon crashing a block of space-time is occupied by the octets and SYN or FIN flags of the last emitted segment, if a new connection is started too soon and uses any of the sequence numbers in the space-time footprint of the last segment of the previous connection incarnation, there is a

potential sequence number overlap area which could cause confusion at the receiver.

### 3.4. Establishing a connection

The "three-way handshake" is the procedure used to establish a connection. This procedure normally is initiated by one TCP and responded to by another TCP. The procedure also works if two TCP simultaneously initiate the procedure. When simultaneous attempt occurs, each TCP receives a "SYN" segment which carries no acknowledgment after it has sent a "SYN". Of course, the arrival of an old duplicate "SYN" segment can potentially make it appear, to the recipient, that a simultaneous connection initiation is in progress. Proper use of "reset" segments can disambiguate these cases.

Several examples of connection initiation follow. Although these examples do not show connection synchronization using data-carrying segments, this is perfectly legitimate, so long as the receiving TCP doesn't deliver the data to the user until it is clear the data is valid (i.e., the data must be buffered at the receiver until the connection reaches the ESTABLISHED state). The three-way handshake reduces the possibility of false connections. It is the implementation of a trade-off between memory and messages to provide information for this checking.

The simplest three-way handshake is shown in Figure 5 below. The figures should be interpreted in the following way. Each line is numbered for reference purposes. Right arrows (-->) indicate departure of a TCP segment from TCP A to TCP B, or arrival of a segment at B from A. Left arrows (<--), indicate the reverse. Ellipsis (...) indicates a segment which is still in the network (delayed). An "XXX" indicates a segment which is lost or rejected. Comments appear in parentheses. TCP states represent the state AFTER the departure or arrival of the segment (whose contents are shown in the center of each line). Segment contents are shown in abbreviated form, with sequence number, control flags, and ACK field. Other fields such as window, addresses, lengths, and text have been left out in the interest of clarity.

TCP A	TCP B
1. CLOSED	LISTEN
2. SYN-SENT --> <SEQ=100><CTL=SYN>	--> SYN-RECEIVED
3. ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
4. ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>	--> ESTABLISHED
5. ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK><DATA>	--> ESTABLISHED

Basic 3-Way Handshake for Connection Synchronization

Figure 5

In line 2 of Figure 5, TCP A begins by sending a SYN segment indicating that it will use sequence numbers starting with sequence number 100. In line 3, TCP B sends a SYN and acknowledges the SYN it received from TCP A. Note that the acknowledgment field indicates TCP B is now expecting to hear sequence 101, acknowledging the SYN which occupied sequence 100.

At line 4, TCP A responds with an empty segment containing an ACK for TCP B's SYN; and in line 5, TCP A sends some data. Note that the sequence number of the segment in line 5 is the same as in line 4 because the ACK does not occupy sequence number space (if it did, we would wind up ACKing ACK's!).

Simultaneous initiation is only slightly more complex, as is shown in Figure 6. Each TCP cycles from CLOSED to SYN-SENT to SYN-RECEIVED to ESTABLISHED.

TCP A		TCP B
1. CLOSED		CLOSED
2. SYN-SENT	--> <SEQ=100><CTL=SYN>	...
3. SYN-RECEIVED	<-- <SEQ=300><CTL=SYN>	<-- SYN-SENT
4.	... <SEQ=100><CTL=SYN>	--> SYN-RECEIVED
5. SYN-RECEIVED	--> <SEQ=100><ACK=301><CTL=SYN,ACK>	...
6. ESTABLISHED	<-- <SEQ=300><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
7.	... <SEQ=100><ACK=301><CTL=SYN,ACK>	--> ESTABLISHED

#### Simultaneous Connection Synchronization

Figure 6

The principle reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion. To deal with this, a special control message, reset, has been devised. If the receiving TCP is in a non-synchronized state (i.e., SYN-SENT, SYN-RECEIVED), it returns to LISTEN on receiving an acceptable reset. If the TCP is in one of the synchronized states (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), it aborts the connection and informs its user. We discuss this latter case under "half-open" connections below.

TCP A		TCP B
1. CLOSED		LISTEN
2. SYN-SENT	--> <SEQ=100><CTL=SYN>	...
3. (duplicate)	... <SEQ=90><CTL=SYN>	--> SYN-RECEIVED
4. SYN-SENT	<-- <SEQ=300><ACK=91><CTL=SYN,ACK>	<-- SYN-RECEIVED
5. SYN-SENT	--> <SEQ=91><CTL=RST>	--> LISTEN
6.	... <SEQ=100><CTL=SYN>	--> SYN-RECEIVED
7. SYN-SENT	<-- <SEQ=400><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
8. ESTABLISHED	--> <SEQ=101><ACK=401><CTL=ACK>	--> ESTABLISHED

## Recovery from Old Duplicate SYN

Figure 7

As a simple example of recovery from old duplicates, consider Figure 7. At line 3, an old duplicate SYN arrives at TCP B. TCP B cannot tell that this is an old duplicate, so it responds normally (line 4). TCP A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ field selected to make the segment believable. TCP B, on receiving the RST, returns to the LISTEN state. When the original SYN (pun intended) finally arrives at line 6, the synchronization proceeds normally. If the SYN at line 6 had arrived before the RST, a more complex exchange might have occurred with RST's sent in both directions.

## Half-Open Connections and Other Anomalies

An established connection is said to be "half-open" if one of the TCPs has closed or aborted the connection at its end without the knowledge of the other, or if the two ends of the connection have become desynchronized owing to a crash that resulted in loss of memory. Such connections will automatically become reset if an attempt is made to send data in either direction. However, half-open connections are expected to be unusual, and the recovery procedure is mildly involved.

If at site A the connection no longer exists, then an attempt by the user at site B to send any data on it will result in the site B TCP receiving a reset control message. Such a message indicates to the

site B TCP that something is wrong, and it is expected to abort the connection.

Assume that two user processes A and B are communicating with one another when a crash occurs causing loss of memory to A's TCP. Depending on the operating system supporting A's TCP, it is likely that some error recovery mechanism exists. When the TCP is up again, A is likely to start again from the beginning or from a recovery point. As a result, A will probably try to OPEN the connection again or try to SEND on the connection it believes open. In the latter case, it receives the error message "connection not open" from the local (A's) TCP. In an attempt to establish the connection, A's TCP will send a segment containing SYN. This scenario leads to the example shown in Figure 8. After TCP A crashes, the user attempts to re-open the connection. TCP B, in the meantime, thinks the connection is open.

TCP A	TCP B
1. (CRASH)	(send 300, receive 100)
2. CLOSED	ESTABLISHED
3. SYN-SENT --> <SEQ=400><CTL=SYN>	--> (??)
4. (!!)	<-- <SEQ=300><ACK=100><CTL=ACK> <-- ESTABLISHED
5. SYN-SENT --> <SEQ=100><CTL=RST>	--> (Abort!!)
6. SYN-SENT	CLOSED
7. SYN-SENT --> <SEQ=400><CTL=SYN>	-->

#### Half-Open Connection Discovery

Figure 8

When the SYN arrives at line 3, TCP B, being in a synchronized state, and the incoming segment outside the window, responds with an acknowledgment indicating what sequence it next expects to hear (ACK 100). TCP A sees that this segment does not acknowledge anything it sent and, being unsynchronized, sends a reset (RST) because it has detected a half-open connection. TCP B aborts at line 5. TCP A will continue to try to establish the connection; the problem is now reduced to the basic 3-way handshake of Figure 5.

An interesting alternative case occurs when TCP A crashes and TCP B tries to send data on what it thinks is a synchronized connection.

This is illustrated in Figure 9. In this case, the data arriving at TCP A from TCP B (line 2) is unacceptable because no such connection exists, so TCP A sends a RST. The RST is acceptable so TCP B processes it and aborts the connection.

TCP A	TCP B
1. (CRASH)	(send 300, receive 100)
2. (??) <-- <SEQ=300><ACK=100><DATA=10><CTL=ACK>	<-- ESTABLISHED
3. --> <SEQ=100><CTL=RST>	--> (ABORT!!)

Active Side Causes Half-Open Connection Discovery

Figure 9

In Figure 10, we find the two TCPs A and B with passive connections waiting for SYN. An old duplicate arriving at TCP B (line 2) stirs B into action. A SYN-ACK is returned (line 3) and causes TCP A to generate a RST (the ACK in line 3 is not acceptable). TCP B accepts the reset and returns to its passive LISTEN state.

TCP A	TCP B
1. LISTEN	LISTEN
2. ... <SEQ=Z><CTL=SYN>	--> SYN-RECEIVED
3. (??) <-- <SEQ=X><ACK=Z+1><CTL=SYN,ACK>	<-- SYN-RECEIVED
4. --> <SEQ=Z+1><CTL=RST>	--> (return to LISTEN!)
5. LISTEN	LISTEN

Old Duplicate SYN Initiates a Reset on two Passive Sockets

Figure 10

A variety of other cases are possible, all of which are accounted for by the following rules for RST generation and processing.

Reset Generation

As a general rule, reset (RST) must be sent whenever a segment arrives which apparently is not intended for the current connection. A reset must not be sent if it is not clear that this is the case.

There are three groups of states:

1. If the connection does not exist (CLOSED) then a reset is sent in response to any incoming segment except another reset. In particular, SYNs addressed to a non-existent connection are rejected by this means.

If the incoming segment has the ACK bit set, the reset takes its sequence number from the ACK field of the segment, otherwise the reset has sequence number zero and the ACK field is set to the sum of the sequence number and segment length of the incoming segment. The connection remains in the CLOSED state.

2. If the connection is in any non-synchronized state (LISTEN, SYN-SENT, SYN-RECEIVED), and the incoming segment acknowledges something not yet sent (the segment carries an unacceptable ACK), or if an incoming segment has a security level or compartment which does not exactly match the level and compartment requested for the connection, a reset is sent.

If our SYN has not been acknowledged and the precedence level of the incoming segment is higher than the precedence level requested then either raise the local precedence level (if allowed by the user and the system) or send a reset; or if the precedence level of the incoming segment is lower than the precedence level requested then continue as if the precedence matched exactly (if the remote TCP cannot raise the precedence level to match ours this will be detected in the next segment it sends, and the connection will be terminated then). If our SYN has been acknowledged (perhaps in this incoming segment) the precedence level of the incoming segment must match the local precedence level exactly, if it does not a reset must be sent.

If the incoming segment has an ACK field, the reset takes its sequence number from the ACK field of the segment, otherwise the reset has sequence number zero and the ACK field is set to the sum of the sequence number and segment length of the incoming segment. The connection remains in the same state.

3. If the connection is in a synchronized state (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), any unacceptable segment (out of window sequence number or unacceptable acknowledgment number) must elicit only an empty acknowledgment segment containing the current send-sequence number

and an acknowledgment indicating the next sequence number expected to be received, and the connection remains in the same state.

If an incoming segment has a security level, or compartment, or precedence which does not exactly match the level, and compartment, and precedence requested for the connection, a reset is sent and the connection goes to the CLOSED state. The reset takes its sequence number from the ACK field of the incoming segment.

#### Reset Processing

In all states except SYN-SENT, all reset (RST) segments are validated by checking their SEQ-fields. A reset is valid if its sequence number is in the window. In the SYN-SENT state (a RST received in response to an initial SYN), the RST is acceptable if the ACK field acknowledges the SYN.

The receiver of a RST first validates it, then changes state. If the receiver was in the LISTEN state, it ignores it. If the receiver was in SYN-RECEIVED state and had previously been in the LISTEN state, then the receiver returns to the LISTEN state, otherwise the receiver aborts the connection and goes to the CLOSED state. If the receiver was in any other state, it aborts the connection and advises the user and goes to the CLOSED state.

### 3.5. Closing a Connection

CLOSE is an operation meaning "I have no more data to send." The notion of closing a full-duplex connection is subject to ambiguous interpretation, of course, since it may not be obvious how to treat the receiving side of the connection. We have chosen to treat CLOSE in a simplex fashion. The user who CLOSEs may continue to RECEIVE until he is told that the other side has CLOSED also. Thus, a program could initiate several SENDs followed by a CLOSE, and then continue to RECEIVE until signaled that a RECEIVE failed because the other side has CLOSED. We assume that the TCP will signal a user, even if no RECEIVES are outstanding, that the other side has closed, so the user can terminate his side gracefully. A TCP will reliably deliver all buffers SENT before the connection was CLOSED so a user who expects no data in return need only wait to hear the connection was CLOSED successfully to know that all his data was received at the destination TCP. Users must keep reading connections they close for sending until the TCP says no more data.

There are essentially three cases:

- 1) The user initiates by telling the TCP to CLOSE the connection

- 2) The remote TCP initiates by sending a FIN control signal
- 3) Both users CLOSE simultaneously

Case 1: Local user initiates the close

In this case, a FIN segment can be constructed and placed on the outgoing segment queue. No further SENDs from the user will be accepted by the TCP, and it enters the FIN-WAIT-1 state. RECEIVES are allowed in this state. All segments preceding and including FIN will be retransmitted until acknowledged. When the other TCP has both acknowledged the FIN and sent a FIN of its own, the first TCP can ACK this FIN. Note that a TCP receiving a FIN will ACK but not send its own FIN until its user has CLOSED the connection also.

Case 2: TCP receives a FIN from the network

If an unsolicited FIN arrives from the network, the receiving TCP can ACK it and tell the user that the connection is closing. The user will respond with a CLOSE, upon which the TCP can send a FIN to the other TCP after sending any remaining data. The TCP then waits until its own FIN is acknowledged whereupon it deletes the connection. If an ACK is not forthcoming, after the user timeout the connection is aborted and the user is told.

Case 3: both users close simultaneously

A simultaneous CLOSE by users at both ends of a connection causes FIN segments to be exchanged. When all segments preceding the FINs have been processed and acknowledged, each TCP can ACK the FIN it has received. Both will, upon receiving these ACKs, delete the connection.

TCP A		TCP B
1. ESTABLISHED		ESTABLISHED
2. (Close)		
FIN-WAIT-1	--> <SEQ=100><ACK=300><CTL=FIN,ACK>	--> CLOSE-WAIT
3. FIN-WAIT-2	<-- <SEQ=300><ACK=101><CTL=ACK>	<-- CLOSE-WAIT
4. TIME-WAIT	<-- <SEQ=300><ACK=101><CTL=FIN,ACK>	(Close) <-- LAST-ACK
5. TIME-WAIT	--> <SEQ=101><ACK=301><CTL=ACK>	--> CLOSED
6. (2 MSL) CLOSED		

## Normal Close Sequence

Figure 11

TCP A		TCP B
1. ESTABLISHED		ESTABLISHED
2. (Close)		(Close)
FIN-WAIT-1	--> <SEQ=100><ACK=300><CTL=FIN,ACK>	... FIN-WAIT-1
	<-- <SEQ=300><ACK=100><CTL=FIN,ACK>	<--
	... <SEQ=100><ACK=300><CTL=FIN,ACK>	-->
3. CLOSING	--> <SEQ=101><ACK=301><CTL=ACK>	... CLOSING
	<-- <SEQ=301><ACK=101><CTL=ACK>	<--
	... <SEQ=101><ACK=301><CTL=ACK>	-->
4. TIME-WAIT (2 MSL) CLOSED		TIME-WAIT (2 MSL) CLOSED

## Simultaneous Close Sequence

Figure 12

## 3.6. Precedence and Security

The intent is that connection be allowed only between ports operating with exactly the same security and compartment values and at the higher of the precedence level requested by the two ports.

The precedence and security parameters used in TCP are exactly those defined in the Internet Protocol (IP) [2]. Throughout this TCP specification the term "security/compartment" is intended to indicate the security parameters used in IP including security, compartment, user group, and handling restriction.

A connection attempt with mismatched security/compartment values or a lower precedence value must be rejected by sending a reset. Rejecting a connection due to too low a precedence only occurs after an acknowledgment of the SYN has been received.

Note that TCP modules which operate only at the default value of precedence will still have to check the precedence of incoming segments and possibly raise the precedence level they use on the connection.

The security parameters may be used even in a non-secure environment (the values would indicate unclassified data), thus hosts in non-secure environments must be prepared to receive the security parameters, though they need not send them.

### 3.7. Segmentation

The term "segmentation" refers to the activity TCP performs when ingesting a stream of bytes from a sending application and packetizing that stream of bytes into TCP segments.

For efficiency and performance reasons, it is desirable to send large segments that contain as many bytes of payload data as possible. However, packets that are too long will either be fragmented or dropped within the network. Some firewalls or middleboxes may drop fragmented packets. In either case, when packets are dropped, the connection can fail; hence, it is best for a TCP implementation to avoid generating fragments.

To enable a TCP sender to maximize the size of segments that it sends, without generating fragments, TCP includes the Maximum Segment Size option to convey endpoint information, and TCP implementations also support Path MTU Discovery to discover the limits and capabilities of intermediate networks.

When TCP is used in a situation where either the IP or TCP headers are not minimum, the sender must reduce the amount of TCP data in any given packet by the number of octets used by the IP and TCP options. The rationale for this is explained in RFC 6691.

### 3.7.1. Maximum Segment Size Option

TCP MUST implement both sending and receiving the Maximum Segment Size option.

TCP SHOULD send an MSS (Maximum Segment Size) option in every SYN segment when its receive MSS differs from the default 536, and MAY send it always.

If an MSS option is not received at connection setup, TCP MUST assume a default send MSS of 536 (576-40).

The maximum size of a segment that TCP really sends, the "effective send MSS," MUST be the smaller of the send MSS (which reflects the available reassembly buffer size at the remote host) and the largest size permitted by the IP layer:

$$\text{Eff.snd.MSS} =$$
$$\min(\text{SendMSS}+20, \text{MMS\_S}) - \text{TCP}h\text{drsize} - \text{IPOptionsize}$$

where:

- o SendMSS is the MSS value received from the remote host, or the default 536 if no MSS option is received.
- o MMS\_S is the maximum size for a transport-layer message that TCP may send.
- o TCPhdrsize is the size of the fixed TCP header; this is normally 20, but may be larger if TCP options are to be sent.
- o IPOptionsize is the size of any IP options that TCP will pass to the IP layer with the current message.

The MSS value to be sent in an MSS option should be equal to the effective MTU minus the fixed IP and TCP headers. By ignoring both IP and TCP options when calculating the value for the MSS option, if there are any IP or TCP options to be sent in a packet, then the sender must decrease the size of the TCP data accordingly. RFC 6691 discusses this in greater detail.

The MSS value to be sent in an MSS option must be less than or equal to:

$$\text{MMS\_R} - 20$$

where MMS\_R is the maximum size for a transport-layer message that can be received (and reassembled). TCP obtains MMS\_R and MMS\_S from the IP layer; see the generic call GET\_MAXSIZES in Section 3.4 of RFC 1122.

### 3.7.2. Path MTU Discovery

The TCP MSS option specifies an upper bound for the size of packets that can be received. Hence, setting the value in the MSS option too small can impact the ability for Path MTU Discovery to find a larger path MTU. For more information on Path MTU Discovery, see:

- o "Path MTU Discovery" [RFC1191]
- o "TCP Problems with Path MTU Discovery" [RFC2923]
- o "Packetization Layer Path MTU Discovery" [RFC4821]

### 3.7.3. Interfaces with Variable MSS Values

The effective MTU can sometimes vary, as when used with variable compression, e.g., RObust Header Compression (ROHC) [RFC5795]. It is tempting for TCP to want to advertise the largest possible MSS, to support the most efficient use of compressed payloads. Unfortunately, some compression schemes occasionally need to transmit full headers (and thus smaller payloads) to resynchronize state at their endpoint compressors/decompressors. If the largest MTU is used to calculate the value to advertise in the MSS option, TCP retransmission may interfere with compressor resynchronization.

As a result, when the effective MTU of an interface varies, TCP SHOULD use the smallest effective MTU of the interface to calculate the value to advertise in the MSS option.

### 3.7.4. IPv6 Jumbograms

In order to support TCP over IPv6 jumbograms, implementations need to be able to send TCP segments larger than 64K. RFC 2675 [RFC2675] defines that a value of 65,535 is to be treated as infinity, and Path MTU Discovery [RFC1981] is used to determine the actual MSS.

## 3.8. Data Communication

Once the connection is established data is communicated by the exchange of segments. Because segments may be lost due to errors (checksum test failure), or network congestion, TCP uses retransmission (after a timeout) to ensure delivery of every segment. Duplicate segments may arrive due to network or TCP retransmission.

As discussed in the section on sequence numbers the TCP performs certain tests on the sequence and acknowledgment numbers in the segments to verify their acceptability.

The sender of data keeps track of the next sequence number to use in the variable SND.NXT. The receiver of data keeps track of the next sequence number to expect in the variable RCV.NXT. The sender of data keeps track of the oldest unacknowledged sequence number in the variable SND.UNA. If the data flow is momentarily idle and all data sent has been acknowledged then the three variables will be equal.

When the sender creates a segment and transmits it the sender advances SND.NXT. When the receiver accepts a segment it advances RCV.NXT and sends an acknowledgment. When the data sender receives an acknowledgment it advances SND.UNA. The extent to which the values of these variables differ is a measure of the delay in the communication. The amount by which the variables are advanced is the length of the data and SYN or FIN flags in the segment. Note that once in the ESTABLISHED state all segments must carry current acknowledgment information.

The CLOSE user call implies a push function, as does the FIN control flag in an incoming segment.

#### Retransmission Timeout

NOTE: TODO this needs to be updated in light of 1122 4.2.2.15 and errata 573; this will be done as part of RFC 1122 incorporation into this document.

Because of the variability of the networks that compose an internetwork system and the wide range of uses of TCP connections the retransmission timeout must be dynamically determined. One procedure for determining a retransmission timeout is given here as an illustration.

#### An Example Retransmission Timeout Procedure

Measure the elapsed time between sending a data octet with a particular sequence number and receiving an acknowledgment that covers that sequence number (segments sent do not have to match segments received). This measured elapsed time is the Round Trip Time (RTT). Next compute a Smoothed Round Trip Time (SRTT) as:

$$\text{SRTT} = (\text{ALPHA} * \text{SRTT}) + ((1-\text{ALPHA}) * \text{RTT})$$

and based on this, compute the retransmission timeout (RTO) as:

$$\text{RTO} = \min[\text{UBOUND}, \max[\text{LBOUND}, (\text{BETA} * \text{SRTT})]]$$

where UBOUND is an upper bound on the timeout (e.g., 1 minute), LBOUND is a lower bound on the timeout (e.g., 1 second), ALPHA is a smoothing factor (e.g., .8 to .9), and BETA is a delay variance factor (e.g., 1.3 to 2.0).

#### The Communication of Urgent Information

As a result of implementation differences and middlebox interactions, new applications SHOULD NOT employ the TCP urgent mechanism. However, TCP implementations MUST still include support for the urgent mechanism. Details can be found in RFC 6093 [7].

The objective of the TCP urgent mechanism is to allow the sending user to stimulate the receiving user to accept some urgent data and to permit the receiving TCP to indicate to the receiving user when all the currently known urgent data has been received by the user.

This mechanism permits a point in the data stream to be designated as the end of urgent information. Whenever this point is in advance of the receive sequence number (RCV.NXT) at the receiving TCP, that TCP must tell the user to go into "urgent mode"; when the receive sequence number catches up to the urgent pointer, the TCP must tell user to go into "normal mode". If the urgent pointer is updated while the user is in "urgent mode", the update will be invisible to the user.

The method employs a urgent field which is carried in all segments transmitted. The URG control flag indicates that the urgent field is meaningful and must be added to the segment sequence number to yield the urgent pointer. The absence of this flag indicates that there is no urgent data outstanding.

To send an urgent indication the user must also send at least one data octet. If the sending user also indicates a push, timely delivery of the urgent information to the destination process is enhanced.

A TCP MUST support a sequence of urgent data of any length. [3]

A TCP MUST inform the application layer asynchronously whenever it receives an Urgent pointer and there was previously no pending urgent data, or whenever the Urgent pointer advances in the data stream. There MUST be a way for the application to learn how much urgent data remains to be read from the connection, or at least to determine whether or not more urgent data remains to be read. [3]

#### Managing the Window

The window sent in each segment indicates the range of sequence numbers the sender of the window (the data receiver) is currently prepared to accept. There is an assumption that this is related to the currently available data buffer space available for this connection.

Indicating a large window encourages transmissions. If more data arrives than can be accepted, it will be discarded. This will result in excessive retransmissions, adding unnecessarily to the load on the network and the TCPs. Indicating a small window may restrict the transmission of data to the point of introducing a round trip delay between each new segment transmitted.

The mechanisms provided allow a TCP to advertise a large window and to subsequently advertise a much smaller window without having accepted that much data. This, so called "shrinking the window," is strongly discouraged. The robustness principle dictates that TCPs will not shrink the window themselves, but will be prepared for such behavior on the part of other TCPs.

The sending TCP must be prepared to accept from the user and send at least one octet of new data even if the send window is zero. The sending TCP must regularly retransmit to the receiving TCP even when the window is zero. Two minutes is recommended for the retransmission interval when the window is zero. This retransmission is essential to guarantee that when either TCP has a zero window the re-opening of the window will be reliably reported to the other.

When the receiving TCP has a zero window and a segment arrives it must still send an acknowledgment showing its next expected sequence number and current window (zero).

The sending TCP packages the data to be transmitted into segments which fit the current window, and may repackage segments on the retransmission queue. Such repackaging is not required, but may be helpful.

In a connection with a one-way data flow, the window information will be carried in acknowledgment segments that all have the same sequence number so there will be no way to reorder them if they arrive out of order. This is not a serious problem, but it will allow the window information to be on occasion temporarily based on old reports from the data receiver. A refinement to avoid this problem is to act on the window information from segments that carry the highest acknowledgment number (that is segments with acknowledgment number equal or greater than the highest previously received).

The window management procedure has significant influence on the communication performance. The following comments are suggestions to implementers.

#### Window Management Suggestions

Allocating a very small window causes data to be transmitted in many small segments when better performance is achieved using fewer large segments.

One suggestion for avoiding small windows is for the receiver to defer updating a window until the additional allocation is at least X percent of the maximum allocation possible for the connection (where X might be 20 to 40).

Another suggestion is for the sender to avoid sending small segments by waiting until the window is large enough before sending data. If the user signals a push function then the data must be sent even if it is a small segment.

Note that the acknowledgments should not be delayed or unnecessary retransmissions will result. One strategy would be to send an acknowledgment when a small segment arrives (with out updating the window information), and then to send another acknowledgment with new window information when the window is larger.

The segment sent to probe a zero window may also begin a break up of transmitted data into smaller and smaller segments. If a segment containing a single data octet sent to probe a zero window is accepted, it consumes one octet of the window now available. If the sending TCP simply sends as much as it can whenever the window is non zero, the transmitted data will be broken into alternating big and small segments. As time goes on, occasional pauses in the receiver making window allocation available will result in breaking the big segments into a small and not quite so big pair. And after a while the data transmission will be in mostly small segments.

The suggestion here is that the TCP implementations need to actively attempt to combine small window allocations into larger windows, since the mechanisms for managing the window tend to lead to many small windows in the simplest minded implementations.

### 3.9. Interfaces

There are of course two interfaces of concern: the user/TCP interface and the TCP/lower-level interface. We have a fairly elaborate model of the user/TCP interface, but the interface to the lower level

protocol module is left unspecified here, since it will be specified in detail by the specification of the lower level protocol. For the case that the lower level is IP we note some of the parameter values that TCPs might use.

### 3.9.1. User/TCP Interface

The following functional description of user commands to the TCP is, at best, fictional, since every operating system will have different facilities. Consequently, we must warn readers that different TCP implementations may have different user interfaces. However, all TCPs must provide a certain minimum set of services to guarantee that all TCP implementations can support the same protocol hierarchy. This section specifies the functional interfaces required of all TCP implementations.

#### TCP User Commands

The following sections functionally characterize a USER/TCP interface. The notation used is similar to most procedure or function calls in high level languages, but this usage is not meant to rule out trap type service calls (e.g., SVCs, UUOs, EMTs).

The user commands described below specify the basic functions the TCP must perform to support interprocess communication. Individual implementations must define their own exact format, and may provide combinations or subsets of the basic functions in single calls. In particular, some implementations may wish to automatically OPEN a connection on the first SEND or RECEIVE issued by the user for a given connection.

In providing interprocess communication facilities, the TCP must not only accept commands, but must also return information to the processes it serves. The latter consists of:

- (a) general information about a connection (e.g., interrupts, remote close, binding of unspecified foreign socket).
- (b) replies to specific user commands indicating success or various types of failure.

#### Open

Format: OPEN (local port, foreign socket, active/passive [, timeout] [, precedence] [, security/compartments] [, options])  
-> local connection name

We assume that the local TCP is aware of the identity of the processes it serves and will check the authority of the process to use the connection specified. Depending upon the implementation of the TCP, the local network and TCP identifiers for the source address will either be supplied by the TCP or the lower level protocol (e.g., IP). These considerations are the result of concern about security, to the extent that no TCP be able to masquerade as another one, and so on. Similarly, no process can masquerade as another without the collusion of the TCP.

If the active/passive flag is set to passive, then this is a call to LISTEN for an incoming connection. A passive open may have either a fully specified foreign socket to wait for a particular connection or an unspecified foreign socket to wait for any call. A fully specified passive call can be made active by the subsequent execution of a SEND.

A transmission control block (TCB) is created and partially filled in with data from the OPEN command parameters.

On an active OPEN command, the TCP will begin the procedure to synchronize (i.e., establish) the connection at once.

The timeout, if present, permits the caller to set up a timeout for all data submitted to TCP. If data is not successfully delivered to the destination within the timeout period, the TCP will abort the connection. The present global default is five minutes.

The TCP or some component of the operating system will verify the users authority to open a connection with the specified precedence or security/compartments. The absence of precedence or security/compartments specification in the OPEN call indicates the default values must be used.

TCP will accept incoming requests as matching only if the security/compartments information is exactly the same and only if the precedence is equal to or higher than the precedence requested in the OPEN call.

The precedence for the connection is the higher of the values requested in the OPEN call and received from the incoming request, and fixed at that value for the life of the connection. Implementers may want to give the user control of this precedence negotiation. For example, the user might be allowed to specify that the precedence must be exactly matched,

or that any attempt to raise the precedence be confirmed by the user.

A local connection name will be returned to the user by the TCP. The local connection name can then be used as a short hand term for the connection defined by the <local socket, foreign socket> pair.

#### Send

Format: SEND (local connection name, buffer address, byte count, PUSH flag, URGENT flag [,timeout])

This call causes the data contained in the indicated user buffer to be sent on the indicated connection. If the connection has not been opened, the SEND is considered an error. Some implementations may allow users to SEND first; in which case, an automatic OPEN would be done. If the calling process is not authorized to use this connection, an error is returned.

If the PUSH flag is set, the data must be transmitted promptly to the receiver, and the PUSH bit will be set in the last TCP segment created from the buffer. If the PUSH flag is not set, the data may be combined with data from subsequent SENDS for transmission efficiency.

New applications SHOULD NOT set the URGENT flag [7] due to implementation differences and middlebox issues.

If the URGENT flag is set, segments sent to the destination TCP will have the urgent pointer set. The receiving TCP will signal the urgent condition to the receiving process if the urgent pointer indicates that data preceding the urgent pointer has not been consumed by the receiving process. The purpose of urgent is to stimulate the receiver to process the urgent data and to indicate to the receiver when all the currently known urgent data has been received. The number of times the sending user's TCP signals urgent will not necessarily be equal to the number of times the receiving user will be notified of the presence of urgent data.

If no foreign socket was specified in the OPEN, but the connection is established (e.g., because a LISTENing connection has become specific due to a foreign segment arriving for the local socket), then the designated buffer is sent to the implied foreign socket. Users who make use of OPEN with an

unspecified foreign socket can make use of SEND without ever explicitly knowing the foreign socket address.

However, if a SEND is attempted before the foreign socket becomes specified, an error will be returned. Users can use the STATUS call to determine the status of the connection. In some implementations the TCP may notify the user when an unspecified socket is bound.

If a timeout is specified, the current user timeout for this connection is changed to the new one.

In the simplest implementation, SEND would not return control to the sending process until either the transmission was complete or the timeout had been exceeded. However, this simple method is both subject to deadlocks (for example, both sides of the connection might try to do SENDs before doing any RECEIVES) and offers poor performance, so it is not recommended. A more sophisticated implementation would return immediately to allow the process to run concurrently with network I/O, and, furthermore, to allow multiple SENDs to be in progress. Multiple SENDs are served in first come, first served order, so the TCP will queue those it cannot service immediately.

We have implicitly assumed an asynchronous user interface in which a SEND later elicits some kind of SIGNAL or pseudo-interrupt from the serving TCP. An alternative is to return a response immediately. For instance, SENDs might return immediate local acknowledgment, even if the segment sent had not been acknowledged by the distant TCP. We could optimistically assume eventual success. If we are wrong, the connection will close anyway due to the timeout. In implementations of this kind (synchronous), there will still be some asynchronous signals, but these will deal with the connection itself, and not with specific segments or buffers.

In order for the process to distinguish among error or success indications for different SENDs, it might be appropriate for the buffer address to be returned along with the coded response to the SEND request. TCP-to-user signals are discussed below, indicating the information which should be returned to the calling process.

#### Receive

Format: RECEIVE (local connection name, buffer address, byte count) -> byte count, urgent flag, push flag

This command allocates a receiving buffer associated with the specified connection. If no OPEN precedes this command or the calling process is not authorized to use this connection, an error is returned.

In the simplest implementation, control would not return to the calling program until either the buffer was filled, or some error occurred, but this scheme is highly subject to deadlocks. A more sophisticated implementation would permit several RECEIVES to be outstanding at once. These would be filled as segments arrive. This strategy permits increased throughput at the cost of a more elaborate scheme (possibly asynchronous) to notify the calling program that a PUSH has been seen or a buffer filled.

If enough data arrive to fill the buffer before a PUSH is seen, the PUSH flag will not be set in the response to the RECEIVE. The buffer will be filled with as much data as it can hold. If a PUSH is seen before the buffer is filled the buffer will be returned partially filled and PUSH indicated.

If there is urgent data the user will have been informed as soon as it arrived via a TCP-to-user signal. The receiving user should thus be in "urgent mode". If the URGENT flag is on, additional urgent data remains. If the URGENT flag is off, this call to RECEIVE has returned all the urgent data, and the user may now leave "urgent mode". Note that data following the urgent pointer (non-urgent data) cannot be delivered to the user in the same buffer with preceding urgent data unless the boundary is clearly marked for the user.

To distinguish among several outstanding RECEIVES and to take care of the case that a buffer is not completely filled, the return code is accompanied by both a buffer pointer and a byte count indicating the actual length of the data received.

Alternative implementations of RECEIVE might have the TCP allocate buffer storage, or the TCP might share a ring buffer with the user.

#### Close

Format: CLOSE (local connection name)

This command causes the connection specified to be closed. If the connection is not open or the calling process is not authorized to use this connection, an error is returned. Closing connections is intended to be a graceful operation in

the sense that outstanding SENDs will be transmitted (and retransmitted), as flow control permits, until all have been serviced. Thus, it should be acceptable to make several SEND calls, followed by a CLOSE, and expect all the data to be sent to the destination. It should also be clear that users should continue to RECEIVE on CLOSING connections, since the other side may be trying to transmit the last of its data. Thus, CLOSE means "I have no more to send" but does not mean "I will not receive any more." It may happen (if the user level protocol is not well thought out) that the closing side is unable to get rid of all its data before timing out. In this event, CLOSE turns into ABORT, and the closing TCP gives up.

The user may CLOSE the connection at any time on his own initiative, or in response to various prompts from the TCP (e.g., remote close executed, transmission timeout exceeded, destination inaccessible).

Because closing a connection requires communication with the foreign TCP, connections may remain in the closing state for a short time. Attempts to reopen the connection before the TCP replies to the CLOSE command will result in error responses.

Close also implies push function.

#### Status

Format: STATUS (local connection name) -> status data

This is an implementation dependent user command and could be excluded without adverse effect. Information returned would typically come from the TCB associated with the connection.

This command returns a data block containing the following information:

- local socket,
- foreign socket,
- local connection name,
- receive window,
- send window,
- connection state,
- number of buffers awaiting acknowledgment,
- number of buffers pending receipt,
- urgent state,
- precedence,
- security/compartments,
- and transmission timeout.

Depending on the state of the connection, or on the implementation itself, some of this information may not be available or meaningful. If the calling process is not authorized to use this connection, an error is returned. This prevents unauthorized processes from gaining information about a connection.

#### Abort

Format: ABORT (local connection name)

This command causes all pending SENDs and RECEIVES to be aborted, the TCB to be removed, and a special RESET message to be sent to the TCP on the other side of the connection. Depending on the implementation, users may receive abort indications for each outstanding SEND or RECEIVE, or may simply receive an ABORT-acknowledgment.

#### TCP-to-User Messages

It is assumed that the operating system environment provides a means for the TCP to asynchronously signal the user program. When the TCP does signal a user program, certain information is passed to the user. Often in the specification the information will be an error message. In other cases there will be information relating to the completion of processing a SEND or RECEIVE or other user call.

The following information is provided:

Local Connection Name	Always
Response String	Always
Buffer Address	Send & Receive
Byte count (counts bytes received)	Receive
Push flag	Receive
Urgent flag	Receive

#### 3.9.2. TCP/Lower-Level Interface

The TCP calls on a lower level protocol module to actually send and receive information over a network. One case is that of the ARPA internetwork system where the lower level module is the Internet Protocol (IP) [2].

If the lower level protocol is IP it provides arguments for a type of service and for a time to live. TCP uses the following settings for these parameters:

Type of Service = Precedence: given by user, Delay: normal, Throughput: normal, Reliability: normal; or binary XXX00000, where XXX are the three bits determining precedence, e.g. 000 means routine precedence.

Time to Live = one minute, or 00111100.

Note that the assumed maximum segment lifetime is two minutes. Here we explicitly ask that a segment be destroyed if it cannot be delivered by the internet system within one minute.

If the lower level is IP (or other protocol that provides this feature) and source routing is used, the interface must allow the route information to be communicated. This is especially important so that the source and destination addresses used in the TCP checksum be the originating source and ultimate destination. It is also important to preserve the return route to answer connection requests.

Any lower level protocol will have to provide the source address, destination address, and protocol fields, and some way to determine the "TCP length", both to provide the functional equivalent service of IP and to be used in the TCP checksum.

### 3.10. Event Processing

The processing depicted in this section is an example of one possible implementation. Other implementations may have slightly different processing sequences, but they should differ from those in this section only in detail, not in substance.

The activity of the TCP can be characterized as responding to events. The events that occur can be cast into three categories: user calls, arriving segments, and timeouts. This section describes the processing the TCP does in response to each of the events. In many cases the processing required depends on the state of the connection.

Events that occur:

#### User Calls

- OPEN
- SEND
- RECEIVE
- CLOSE
- ABORT
- STATUS

#### Arriving Segments

## SEGMENT ARRIVES

## Timeouts

USER TIMEOUT  
RETRANSMISSION TIMEOUT  
TIME-WAIT TIMEOUT

The model of the TCP/user interface is that user commands receive an immediate return and possibly a delayed response via an event or pseudo interrupt. In the following descriptions, the term "signal" means cause a delayed response.

Error responses are given as character strings. For example, user commands referencing connections that do not exist receive "error: connection not open".

Please note in the following that all arithmetic on sequence numbers, acknowledgment numbers, windows, et cetera, is modulo  $2^{32}$  the size of the sequence number space. Also note that " $=<$ " means less than or equal to (modulo  $2^{32}$ ).

A natural way to think about processing incoming segments is to imagine that they are first tested for proper sequence number (i.e., that their contents lie in the range of the expected "receive window" in the sequence number space) and then that they are generally queued and processed in sequence number order.

When a segment overlaps other already received segments we reconstruct the segment to contain just the new data, and adjust the header fields to be consistent.

Note that if no state change is mentioned the TCP stays in the same state.

## OPEN Call

CLOSED STATE (i.e., TCB does not exist)

Create a new transmission control block (TCB) to hold connection state information. Fill in local socket identifier, foreign socket, precedence, security/compartments, and user timeout information. Note that some parts of the foreign socket may be unspecified in a passive OPEN and are to be filled in by the parameters of the incoming SYN segment. Verify the security and precedence requested are allowed for this user, if not return "error: precedence not allowed" or "error: security/compartments not allowed." If passive enter the LISTEN state and return. If active and the foreign socket is unspecified, return "error: foreign socket unspecified"; if active and the foreign socket is specified, issue a SYN segment. An initial send sequence number (ISS) is selected. A SYN segment of the form <SEQ=ISS><CTL=SYN> is sent. Set SND.UNA to ISS, SND.NXT to ISS+1, enter SYN-SENT state, and return.

If the caller does not have access to the local socket specified, return "error: connection illegal for this process". If there is no room to create a new connection, return "error: insufficient resources".

## LISTEN STATE

If active and the foreign socket is specified, then change the connection from passive to active, select an ISS. Send a SYN segment, set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. The urgent bit if requested in the command must be sent with the data segments sent as a result of this command. If there is no room to queue the request, respond with "error: insufficient resources". If Foreign socket was not specified, then return "error: foreign socket unspecified".

SYN-SENT STATE  
SYN-RECEIVED STATE  
ESTABLISHED STATE  
FIN-WAIT-1 STATE  
FIN-WAIT-2 STATE  
CLOSE-WAIT STATE  
CLOSING STATE  
LAST-ACK STATE  
TIME-WAIT STATE

Return "error: connection already exists".

## SEND Call

CLOSED STATE (i.e., TCB does not exist)

If the user does not have access to such a connection, then return "error: connection illegal for this process".

Otherwise, return "error: connection does not exist".

LISTEN STATE

If the foreign socket is specified, then change the connection from passive to active, select an ISS. Send a SYN segment, set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. The urgent bit if requested in the command must be sent with the data segments sent as a result of this command. If there is no room to queue the request, respond with "error: insufficient resources". If Foreign socket was not specified, then return "error: foreign socket unspecified".

SYN-SENT STATE

SYN-RECEIVED STATE

Queue the data for transmission after entering ESTABLISHED state. If no space to queue, respond with "error: insufficient resources".

ESTABLISHED STATE

CLOSE-WAIT STATE

Segmentize the buffer and send it with a piggybacked acknowledgment (acknowledgment value = RCV.NXT). If there is insufficient space to remember this buffer, simply return "error: insufficient resources".

If the urgent flag is set, then  $\text{SND.UP} \leftarrow \text{SND.NXT}$  and set the urgent pointer in the outgoing segments.

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

Return "error: connection closing" and do not service request.

## RECEIVE Call

CLOSED STATE (i.e., TCB does not exist)

If the user does not have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

SYN-SENT STATE

SYN-RECEIVED STATE

Queue for processing after entering ESTABLISHED state. If there is no room to queue this request, respond with "error: insufficient resources".

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

If insufficient incoming segments are queued to satisfy the request, queue the request. If there is no queue space to remember the RECEIVE, respond with "error: insufficient resources".

Reassemble queued incoming segments into receive buffer and return to user. Mark "push seen" (PUSH) if this is the case.

If RCV.UP is in advance of the data currently being passed to the user notify the user of the presence of urgent data.

When the TCP takes responsibility for delivering data to the user that fact must be communicated to the sender via an acknowledgment. The formation of such an acknowledgment is described below in the discussion of processing an incoming segment.

CLOSE-WAIT STATE

Since the remote side has already sent FIN, RECEIVES must be satisfied by text already on hand, but not yet delivered to the user. If no text is awaiting delivery, the RECEIVE will get a "error: connection closing" response. Otherwise, any remaining text can be used to satisfy the RECEIVE.

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

Return "error: connection closing".

## CLOSE Call

CLOSED STATE (i.e., TCB does not exist)

If the user does not have access to such a connection, return "error: connection illegal for this process".

Otherwise, return "error: connection does not exist".

LISTEN STATE

Any outstanding RECEIVES are returned with "error: closing" responses. Delete TCB, enter CLOSED state, and return.

SYN-SENT STATE

Delete the TCB and return "error: closing" responses to any queued SENDs, or RECEIVES.

SYN-RECEIVED STATE

If no SENDs have been issued and there is no pending data to send, then form a FIN segment and send it, and enter FIN-WAIT-1 state; otherwise queue for processing after entering ESTABLISHED state.

ESTABLISHED STATE

Queue this until all preceding SENDs have been segmentized, then form a FIN segment and send it. In any case, enter FIN-WAIT-1 state.

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

Strictly speaking, this is an error and should receive a "error: connection closing" response. An "ok" response would be acceptable, too, as long as a second FIN is not emitted (the first FIN may be retransmitted though).

CLOSE-WAIT STATE

Queue this request until all preceding SENDs have been segmentized; then send a FIN segment, enter LAST-ACK state.

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

Respond with "error: connection closing".

## ABORT Call

CLOSED STATE (i.e., TCB does not exist)

If the user should not have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

## LISTEN STATE

Any outstanding RECEIVES should be returned with "error: connection reset" responses. Delete TCB, enter CLOSED state, and return.

## SYN-SENT STATE

All queued SENDS and RECEIVES should be given "connection reset" notification, delete the TCB, enter CLOSED state, and return.

## SYN-RECEIVED STATE

## ESTABLISHED STATE

## FIN-WAIT-1 STATE

## FIN-WAIT-2 STATE

## CLOSE-WAIT STATE

Send a reset segment:

<SEQ=SND.NXT><CTL=RST>

All queued SENDS and RECEIVES should be given "connection reset" notification; all segments queued for transmission (except for the RST formed above) or retransmission should be flushed, delete the TCB, enter CLOSED state, and return.

## CLOSING STATE LAST-ACK STATE TIME-WAIT STATE

Respond with "ok" and delete the TCB, enter CLOSED state, and return.

## STATUS Call

CLOSED STATE (i.e., TCB does not exist)

If the user should not have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Return "state = LISTEN", and the TCB pointer.

SYN-SENT STATE

Return "state = SYN-SENT", and the TCB pointer.

SYN-RECEIVED STATE

Return "state = SYN-RECEIVED", and the TCB pointer.

ESTABLISHED STATE

Return "state = ESTABLISHED", and the TCB pointer.

FIN-WAIT-1 STATE

Return "state = FIN-WAIT-1", and the TCB pointer.

FIN-WAIT-2 STATE

Return "state = FIN-WAIT-2", and the TCB pointer.

CLOSE-WAIT STATE

Return "state = CLOSE-WAIT", and the TCB pointer.

CLOSING STATE

Return "state = CLOSING", and the TCB pointer.

LAST-ACK STATE

Return "state = LAST-ACK", and the TCB pointer.

TIME-WAIT STATE

Return "state = TIME-WAIT", and the TCB pointer.

## SEGMENT ARRIVES

If the state is CLOSED (i.e., TCB does not exist) then

all data in the incoming segment is discarded. An incoming segment containing a RST is discarded. An incoming segment not containing a RST causes a RST to be sent in response. The acknowledgment and sequence field values are selected to make the reset sequence acceptable to the TCP that sent the offending segment.

If the ACK bit is off, sequence number zero is used,

<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>

If the ACK bit is on,

<SEQ=SEG.ACK><CTL=RST>

Return.

If the state is LISTEN then

first check for an RST

An incoming RST should be ignored. Return.

second check for an ACK

Any acknowledgment is bad if it arrives on a connection still in the LISTEN state. An acceptable reset segment should be formed for any arriving ACK-bearing segment. The RST should be formatted as follows:

<SEQ=SEG.ACK><CTL=RST>

Return.

third check for a SYN

If the SYN bit is set, check the security. If the security/compartment on the incoming segment does not exactly match the security/compartment in the TCB then send a reset and return.

<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>

If the SEG.PRC is greater than the TCB.PRC then if allowed by the user and the system set TCB.PRC<-SEG.PRC, if not allowed send a reset and return.

<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>

If the SEG.PRC is less than the TCB.PRC then continue.

Set RCV.NXT to SEG.SEQ+1, IRS is set to SEG.SEQ and any other control or text should be queued for processing later. ISS should be selected and a SYN segment sent of the form:

<SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

SND.NXT is set to ISS+1 and SND.UNA to ISS. The connection state should be changed to SYN-RECEIVED. Note that any other incoming control or data (combined with SYN) will be processed in the SYN-RECEIVED state, but processing of SYN and ACK should not be repeated. If the listen was not fully specified (i.e., the foreign socket was not fully specified), then the unspecified fields should be filled in now.

fourth other text or control

Any other control or text-bearing segment (not containing SYN) must have an ACK and thus would be discarded by the ACK processing. An incoming RST segment could not be valid, since it could not have been sent in response to anything sent by this incarnation of the connection. So you are unlikely to get here, but if you do, drop the segment, and return.

If the state is SYN-SENT then

first check the ACK bit

If the ACK bit is set

If SEG.ACK =< ISS, or SEG.ACK > SND.NXT, send a reset (unless the RST bit is set, if so drop the segment and return)

<SEQ=SEG.ACK><CTL=RST>

and discard the segment. Return.

If `SND.UNA < SEG.ACK =< SND.NXT` then the ACK is acceptable. (TODO: in processing Errata ID 3300, it was noted that some stacks in the wild that do not send data on the SYN are just checking that `SEG.ACK == SND.NXT ...` think about whether anything should be said about that here)

second check the RST bit

If the RST bit is set

If the ACK was acceptable then signal the user "error: connection reset", drop the segment, enter CLOSED state, delete TCB, and return. Otherwise (no ACK) drop the segment and return.

third check the security and precedence

If the security/compartments in the segment does not exactly match the security/compartments in the TCB, send a reset

If there is an ACK

`<SEQ=SEG.ACK><CTL=RST>`

Otherwise

`<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>`

If there is an ACK

The precedence in the segment must match the precedence in the TCB, if not, send a reset

`<SEQ=SEG.ACK><CTL=RST>`

If there is no ACK

If the precedence in the segment is higher than the precedence in the TCB then if allowed by the user and the system raise the precedence in the TCB to that in the segment, if not allowed to raise the prec then send a reset.

`<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>`

If the precedence in the segment is lower than the precedence in the TCB continue.

If a reset was sent, discard the segment and return.

fourth check the SYN bit

This step should be reached only if the ACK is ok, or there is no ACK, and if the segment did not contain a RST.

If the SYN bit is on and the security/compartments and precedence are acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to SEG.SEQ. SND.UNA should be advanced to equal SEG.ACK (if there is an ACK), and any segments on the retransmission queue which are thereby acknowledged should be removed.

If SND.UNA > ISS (our SYN has been ACKed), change the connection state to ESTABLISHED, form an ACK segment

```
<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>
```

and send it. Data or controls which were queued for transmission may be included. If there are other controls or text in the segment then continue processing at the sixth step below where the URG bit is checked, otherwise return.

Otherwise enter SYN-RECEIVED, form a SYN,ACK segment

```
<SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>
```

and send it. Set the variables:

```
SND.WND <- SEG.WND  
SND.WL1 <- SEG.SEQ  
SND.WL2 <- SEG.ACK
```

If there are other controls or text in the segment, queue them for processing after the ESTABLISHED state has been reached, return.

fifth, if neither of the SYN or RST bits is set then drop the segment and return.

Otherwise,

first check sequence number

```
SYN-RECEIVED STATE  
ESTABLISHED STATE  
FIN-WAIT-1 STATE
```

FIN-WAIT-2 STATE  
 CLOSE-WAIT STATE  
 CLOSING STATE  
 LAST-ACK STATE  
 TIME-WAIT STATE

Segments are processed in sequence. Initial tests on arrival are used to discard old duplicates, but further processing is done in SEG.SEQ order. If a segment's contents straddle the boundary between old and new, only the new parts should be processed.

There are four cases for the acceptability test for an incoming segment:

Segment Length	Receive Window	Test
0	0	SEG.SEQ = RCV.NXT
0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
>0	0	not acceptable
>0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND or RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND

If the RCV.WND is zero, no segments will be acceptable, but special allowance should be made to accept valid ACKs, URGs and RSTs.

If an incoming segment is not acceptable, an acknowledgment should be sent in reply (unless the RST bit is set, if so drop the segment and return):

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

After sending the acknowledgment, drop the unacceptable segment and return.

In the following it is assumed that the segment is the idealized segment that begins at RCV.NXT and does not exceed the window. One could tailor actual segments to fit this assumption by trimming off any portions that lie outside the window (including SYN and FIN), and only processing further

if the segment then begins at RCV.NXT. Segments with higher beginning sequence numbers should be held for later processing.

second check the RST bit,

#### SYN-RECEIVED STATE

If the RST bit is set

If this connection was initiated with a passive OPEN (i.e., came from the LISTEN state), then return this connection to LISTEN state and return. The user need not be informed. If this connection was initiated with an active OPEN (i.e., came from SYN-SENT state) then the connection was refused, signal the user "connection refused". In either case, all segments on the retransmission queue should be removed. And in the active OPEN case, enter the CLOSED state and delete the TCB, and return.

#### ESTABLISHED

FIN-WAIT-1

FIN-WAIT-2

CLOSE-WAIT

If the RST bit is set then, any outstanding RECEIVES and SEND should receive "reset" responses. All segment queues should be flushed. Users should also receive an unsolicited general "connection reset" signal. Enter the CLOSED state, delete the TCB, and return.

#### CLOSING STATE

LAST-ACK STATE

TIME-WAIT

If the RST bit is set then, enter the CLOSED state, delete the TCB, and return.

third check security and precedence

#### SYN-RECEIVED

If the security/compartments and precedence in the segment do not exactly match the security/compartments and precedence in the TCB then send a reset, and return.

ESTABLISHED  
FIN-WAIT-1  
FIN-WAIT-2  
CLOSE-WAIT  
CLOSING  
LAST-ACK  
TIME-WAIT

If the security/compartments and precedence in the segment do not exactly match the security/compartments and precedence in the TCB then send a reset, any outstanding RECEIVES and SEND should receive "reset" responses. All segment queues should be flushed. Users should also receive an unsolicited general "connection reset" signal. Enter the CLOSED state, delete the TCB, and return.

Note this check is placed following the sequence check to prevent a segment from an old connection between these ports with a different security or precedence from causing an abort of the current connection.

fourth, check the SYN bit,

SYN-RECEIVED  
ESTABLISHED STATE  
FIN-WAIT STATE-1  
FIN-WAIT STATE-2  
CLOSE-WAIT STATE  
CLOSING STATE  
LAST-ACK STATE  
TIME-WAIT STATE

TODO: need to incorporate RFC 1122 4.2.2.20(e) here

If the SYN is in the window it is an error, send a reset, any outstanding RECEIVES and SEND should receive "reset" responses, all segment queues should be flushed, the user should also receive an unsolicited general "connection reset" signal, enter the CLOSED state, delete the TCB, and return.

If the SYN is not in the window this step would not be reached and an ack would have been sent in the first step (sequence number check).

fifth check the ACK field,

if the ACK bit is off drop the segment and return

if the ACK bit is on

#### SYN-RECEIVED STATE

If  $\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$  then enter ESTABLISHED state and continue processing with variables below set to:

```
SND.WND <- SEG.WND
SND.WL1 <- SEG.SEQ
SND.WL2 <- SEG.ACK
```

If the segment acknowledgment is not acceptable, form a reset segment,

```
<SEQ=SEG.ACK><CTL=RST>
```

and send it.

#### ESTABLISHED STATE

If  $\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$  then, set  $\text{SND.UNA} \leftarrow \text{SEG.ACK}$ . Any segments on the retransmission queue which are thereby entirely acknowledged are removed. Users should receive positive acknowledgments for buffers which have been SENT and fully acknowledged (i.e., SEND buffer should be returned with "ok" response). If the ACK is a duplicate ( $\text{SEG.ACK} \leq \text{SND.UNA}$ ), it can be ignored. If the ACK acks something not yet sent ( $\text{SEG.ACK} > \text{SND.NXT}$ ) then send an ACK, drop the segment, and return.

If  $\text{SND.UNA} \leq \text{SEG.ACK} \leq \text{SND.NXT}$ , the send window should be updated. If ( $\text{SND.WL1} < \text{SEG.SEQ}$  or ( $\text{SND.WL1} = \text{SEG.SEQ}$  and  $\text{SND.WL2} \leq \text{SEG.ACK}$ )), set  $\text{SND.WND} \leftarrow \text{SEG.WND}$ , set  $\text{SND.WL1} \leftarrow \text{SEG.SEQ}$ , and set  $\text{SND.WL2} \leftarrow \text{SEG.ACK}$ .

Note that  $\text{SND.WND}$  is an offset from  $\text{SND.UNA}$ , that  $\text{SND.WL1}$  records the sequence number of the last segment used to update  $\text{SND.WND}$ , and that  $\text{SND.WL2}$  records the acknowledgment number of the last segment used to update  $\text{SND.WND}$ . The check here prevents using old segments to update the window.

#### FIN-WAIT-1 STATE

In addition to the processing for the ESTABLISHED state, if our FIN is now acknowledged then enter FIN-WAIT-2 and continue processing in that state.

#### FIN-WAIT-2 STATE

In addition to the processing for the ESTABLISHED state, if the retransmission queue is empty, the user's CLOSE can be acknowledged ("ok") but do not delete the TCB.

#### CLOSE-WAIT STATE

Do the same processing as for the ESTABLISHED state.

#### CLOSING STATE

In addition to the processing for the ESTABLISHED state, if the ACK acknowledges our FIN then enter the TIME-WAIT state, otherwise ignore the segment.

#### LAST-ACK STATE

The only thing that can arrive in this state is an acknowledgment of our FIN. If our FIN is now acknowledged, delete the TCB, enter the CLOSED state, and return.

#### TIME-WAIT STATE

The only thing that can arrive in this state is a retransmission of the remote FIN. Acknowledge it, and restart the 2 MSL timeout.

sixth, check the URG bit,

ESTABLISHED STATE  
FIN-WAIT-1 STATE  
FIN-WAIT-2 STATE

If the URG bit is set,  $RCV.UP \leftarrow \max(RCV.UP, SEG.UP)$ , and signal the user that the remote side has urgent data if the urgent pointer (RCV.UP) is in advance of the data consumed. If the user has already been signaled (or is still in the "urgent mode") for this continuous sequence of urgent data, do not signal the user again.

CLOSE-WAIT STATE

CLOSING STATE  
LAST-ACK STATE  
TIME-WAIT

This should not occur, since a FIN has been received from the remote side. Ignore the URG.

seventh, process the segment text,

ESTABLISHED STATE  
FIN-WAIT-1 STATE  
FIN-WAIT-2 STATE

Once in the ESTABLISHED state, it is possible to deliver segment text to user RECEIVE buffers. Text from segments can be moved into buffers until either the buffer is full or the segment is empty. If the segment empties and carries an PUSH flag, then the user is informed, when the buffer is returned, that a PUSH has been received.

When the TCP takes responsibility for delivering the data to the user it must also acknowledge the receipt of the data.

Once the TCP takes responsibility for the data it advances RCV.NXT over the data accepted, and adjusts RCV.WND as appropriate to the current buffer availability. The total of RCV.NXT and RCV.WND should not be reduced.

Please note the window management suggestions in section 3.7.

Send an acknowledgment of the form:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

This acknowledgment should be piggybacked on a segment being transmitted if possible without incurring undue delay.

CLOSE-WAIT STATE  
CLOSING STATE  
LAST-ACK STATE  
TIME-WAIT STATE

This should not occur, since a FIN has been received from the remote side. Ignore the segment text.

eighth, check the FIN bit,

Do not process the FIN if the state is CLOSED, LISTEN or SYN-SENT since the SEG.SEQ cannot be validated; drop the segment and return.

If the FIN bit is set, signal the user "connection closing" and return any pending RECEIVES with same message, advance RCV.NXT over the FIN, and send an acknowledgment for the FIN. Note that FIN implies PUSH for any segment text not yet delivered to the user.

SYN-RECEIVED STATE  
ESTABLISHED STATE

Enter the CLOSE-WAIT state.

FIN-WAIT-1 STATE

If our FIN has been ACKed (perhaps in this segment), then enter TIME-WAIT, start the time-wait timer, turn off the other timers; otherwise enter the CLOSING state.

FIN-WAIT-2 STATE

Enter the TIME-WAIT state. Start the time-wait timer, turn off the other timers.

CLOSE-WAIT STATE

Remain in the CLOSE-WAIT state.

CLOSING STATE

Remain in the CLOSING state.

LAST-ACK STATE

Remain in the LAST-ACK state.

TIME-WAIT STATE

Remain in the TIME-WAIT state. Restart the 2 MSL time-wait timeout.

and return.

## USER TIMEOUT

## USER TIMEOUT

For any state if the user timeout expires, flush all queues, signal the user "error: connection aborted due to user timeout" in general and for any outstanding calls, delete the TCB, enter the CLOSED state and return.

## RETRANSMISSION TIMEOUT

For any state if the retransmission timeout expires on a segment in the retransmission queue, send the segment at the front of the retransmission queue again, reinitialize the retransmission timer, and return.

## TIME-WAIT TIMEOUT

If the time-wait timeout expires on a connection delete the TCB, enter the CLOSED state and return.

## 3.11. Glossary

1822 BBN Report 1822, "The Specification of the Interconnection of a Host and an IMP". The specification of interface between a host and the ARPANET.

## ACK

A control bit (acknowledge) occupying no sequence space, which indicates that the acknowledgment field of this segment specifies the next sequence number the sender of this segment is expecting to receive, hence acknowledging receipt of all previous sequence numbers.

## ARPANET message

The unit of transmission between a host and an IMP in the ARPANET. The maximum size is about 1012 octets (8096 bits).

## ARPANET packet

A unit of transmission used internally in the ARPANET between IMPs. The maximum size is about 126 octets (1008 bits).

## connection

A logical communication path identified by a pair of sockets.

## datagram

A message sent in a packet switched computer communications network.

## Destination Address

The destination address, usually the network and host identifiers.

## FIN

A control bit (finis) occupying one sequence number, which indicates that the sender will send no more data or control occupying sequence space.

## fragment

A portion of a logical unit of data, in particular an internet fragment is a portion of an internet datagram.

## FTP

A file transfer protocol.

## header

Control information at the beginning of a message, segment, fragment, packet or block of data.

- host**  
A computer. In particular a source or destination of messages from the point of view of the communication network.
- Identification**  
An Internet Protocol field. This identifying value assigned by the sender aids in assembling the fragments of a datagram.
- IMP**  
The Interface Message Processor, the packet switch of the ARPANET.
- internet address**  
A source or destination address specific to the host level.
- internet datagram**  
The unit of data exchanged between an internet module and the higher level protocol together with the internet header.
- internet fragment**  
A portion of the data of an internet datagram with an internet header.
- IP**  
Internet Protocol.
- IRS**  
The Initial Receive Sequence number. The first sequence number used by the sender on a connection.
- ISN**  
The Initial Sequence Number. The first sequence number used on a connection, (either ISS or IRS). Selected in a way that is unique within a given period of time and is unpredictable to attackers.
- ISS**  
The Initial Send Sequence number. The first sequence number used by the sender on a connection.
- leader**  
Control information at the beginning of a message or block of data. In particular, in the ARPANET, the control information on an ARPANET message at the host-IMP interface.
- left sequence**  
This is the next sequence number to be acknowledged by the data receiving TCP (or the lowest currently unacknowledged

sequence number) and is sometimes referred to as the left edge of the send window.

local packet

The unit of transmission within a local network.

module

An implementation, usually in software, of a protocol or other procedure.

MSL

Maximum Segment Lifetime, the time a TCP segment can exist in the internetwork system. Arbitrarily defined to be 2 minutes.

octet

An eight bit byte.

Options

An Option field may contain several options, and each option may be several octets in length. The options are used primarily in testing situations; for example, to carry timestamps. Both the Internet Protocol and TCP provide for options fields.

packet

A package of data with a header which may or may not be logically complete. More often a physical packaging than a logical packaging of data.

port

The portion of a socket that specifies which logical input or output channel of a process is associated with the data.

process

A program in execution. A source or destination of data from the point of view of the TCP or other host-to-host protocol.

PUSH

A control bit occupying no sequence space, indicating that this segment contains data that must be pushed through to the receiving user.

RCV.NXT

receive next sequence number

RCV.UP

receive urgent pointer

RCV.WND  
receive window

receive next sequence number  
This is the next sequence number the local TCP is expecting to receive.

receive window  
This represents the sequence numbers the local (receiving) TCP is willing to receive. Thus, the local TCP considers that segments overlapping the range RCV.NXT to RCV.NXT + RCV.WND - 1 carry acceptable data or control. Segments containing sequence numbers entirely outside of this range are considered duplicates and discarded.

RST  
A control bit (reset), occupying no sequence space, indicating that the receiver should delete the connection without further interaction. The receiver can determine, based on the sequence number and acknowledgment fields of the incoming segment, whether it should honor the reset command or ignore it. In no case does receipt of a segment containing RST give rise to a RST in response.

RTP  
Real Time Protocol: A host-to-host protocol for communication of time critical information.

SEG.ACK  
segment acknowledgment

SEG.LEN  
segment length

SEG.PRC  
segment precedence value

SEG.SEQ  
segment sequence

SEG.UP  
segment urgent pointer field

SEG.WND  
segment window field

segment

A logical unit of data, in particular a TCP segment is the unit of data transferred between a pair of TCP modules.

segment acknowledgment

The sequence number in the acknowledgment field of the arriving segment.

segment length

The amount of sequence number space occupied by a segment, including any controls which occupy sequence space.

segment sequence

The number in the sequence field of the arriving segment.

send sequence

This is the next sequence number the local (sending) TCP will use on the connection. It is initially selected from an initial sequence number curve (ISN) and is incremented for each octet of data or sequenced control transmitted.

send window

This represents the sequence numbers which the remote (receiving) TCP is willing to receive. It is the value of the window field specified in segments from the remote (data receiving) TCP. The range of new sequence numbers which may be emitted by a TCP lies between SND.NXT and SND.UNA + SND.WND - 1. (Retransmissions of sequence numbers between SND.UNA and SND.NXT are expected, of course.)

SND.NXT

send sequence

SND.UNA

left sequence

SND.UP

send urgent pointer

SND.WL1

segment sequence number at last window update

SND.WL2

segment acknowledgment number at last window update

SND.WND

send window

socket

An address which specifically includes a port identifier, that is, the concatenation of an Internet Address with a TCP port.

Source Address

The source address, usually the network and host identifiers.

SYN

A control bit in the incoming segment, occupying one sequence number, used at the initiation of a connection, to indicate where the sequence numbering will start.

TCB

Transmission control block, the data structure that records the state of a connection.

TCB.PRC

The precedence of the connection.

TCP

Transmission Control Protocol: A host-to-host protocol for reliable communication in internetwork environments.

TOS

Type of Service, an Internet Protocol field.

Type of Service

An Internet Protocol field which indicates the type of service for this internet fragment.

URG

A control bit (urgent), occupying no sequence space, used to indicate that the receiving user should be notified to do urgent processing as long as there is data to be consumed with sequence numbers less than the value indicated in the urgent pointer.

urgent pointer

A control field meaningful only when the URG bit is on. This field communicates the value of the urgent pointer which indicates the data octet associated with the sending user's urgent call.

#### 4. Changes from RFC 793

This document obsoletes RFC 793 as well as RFC 6093 and 6528, which updated 793. In all cases, only the normative protocol specification and requirements have been incorporated into this document, and the

informational text with background and rationale has not been carried in. The informational content of those documents is still valuable in learning about and understanding TCP, and they are valid Informational references, even though their normative content has been incorporated into this document.

The main body of this document was adapted from RFC 793's Section 3, titled "FUNCTIONAL SPECIFICATION", with an attempt to keep formatting and layout as close as possible.

The collection of applicable RFC Errata that have been reported and either accepted or held for an update to RFC 793 were incorporated (Errata IDs: 573, 574, 700, 701, 1283, 1561, 1562, 1564, 1565, 1571, 1572, 2296, 2297, 2298, 2748, 2749, 2934, 3213, 3300, 3301). Some errata were not applicable due to other changes (Errata IDs: 572, 575, 1569, 3602). TODO: 3305

Changes to the specification of the Urgent Pointer described in RFC 1122 and 6093 were incorporated. See RFC 6093 for detailed discussion of why these changes were necessary.

The more secure Initial Sequence Number generation algorithm from RFC 6528 was incorporated. See RFC 6528 for discussion of the attacks that this mitigates, as well as advice on selecting PRF algorithms and managing secret key data.

RFC EDITOR'S NOTE: the content below is for detailed change tracking and planning, and not to be included with the final revision of the document.

The -00 revision of this document was merely a proposal and rough plan for updating RFC 793.

The -01 revision of this document incorporates the content of RFC 793 Section 3 titled "FUNCTIONAL SPECIFICATION". Other content from RFC 793 has not been incorporated. The -01 revision of this document makes some minor formatting changes to the RFC 793 content in order to convert the content into XML2RFC format and account for left-out parts of RFC 793. For instance, figure numbering differs and some indentation is not exactly the same.

The -02 revision of this document incorporates errata that have been verified:

Errata ID 573: Reported by Bob Braden (note: This errata basically is just a reminder that RFC 1122 updates 793. Some of the associated changes are left pending to a separate revision that incorporates 1122. Bob's mention of PUSH in 793 section 2.8 was

not applicable here because that section was not part of the "functional specification". Also the 1122 text on the retransmission timeout also has been updated by subsequent RFCs, so the change here deviates from Bob's suggestion to apply the 1122 text.)

Errata ID 574: Reported by Yin Shuming

Errata ID 700: Reported by Yin Shuming

Errata ID 701: Reported by Yin Shuming

Errata ID 1283: Reported by Pei-chun Cheng

Errata ID 1561: Reported by Constantin Hagemeier

Errata ID 1562: Reported by Constantin Hagemeier

Errata ID 1564: Reported by Constantin Hagemeier

Errata ID 1565: Reported by Constantin Hagemeier

Errata ID 1571: Reported by Constantin Hagemeier

Errata ID 1572: Reported by Constantin Hagemeier

Errata ID 2296: Reported by Vishwas Manral

Errata ID 2297: Reported by Vishwas Manral

Errata ID 2298: Reported by Vishwas Manral

Errata ID 2748: Reported by Mykyta Yevstifeyev

Errata ID 2749: Reported by Mykyta Yevstifeyev

Errata ID 2934: Reported by Constantin Hagemeier

Errata ID 3213: Reported by EugnJun Yi

Errata ID 3300: Reported by Botong Huang

Errata ID 3301: Reported by Botong Huang

Note: Some verified errata were not used in this update, as they relate to sections of RFC 793 elided from this document. These include Errata ID 572, 575, and 1569.

Note: Errata ID 3602 was not applied in this revision as it is duplicative of the 1122 corrections.

There is an errata 3305 currently reported that need to be verified, held, or rejected by the ADs; it is addressing the same issue as draft-gont-tcpm-tcp-seq-validation and was not attempted to be applied to this document.

Not related to RFC 793 content, this revision also makes small tweaks to the introductory text, fixes indentation of the pseudoheader diagram, and notes that the Security Considerations should also include privacy, when this section is written.

The -03 revision of this document revises all discussion of the urgent pointer in order to comply with RFC 6093, 1122, and 1011. Since 1122 held requirements on the urgent pointer, the full list of requirements was brought into an appendix of this document, so that it can be updated as-needed.

The -04 revision of this document includes the ISN generation changes from RFC 6528.

The -05 revision of this document incorporates MSS requirements and definitions from RFC 879, 1122, and 6691, as well as option-handling requirements from RFC 1122.

TODO: Incomplete list of planned changes - these need to be added to and made more specific, as the document proceeds:

1. incorporate 1122 additions
  2. point to major additional docs like 1323bis and 5681
  3. incorporate relevant parts of 3168 (ECN)
  4. incorporate Fernando's new number-checking fixes (if past the IESG in time)
  5. point to PMTUD?
  6. point to 5461 (soft errors)
  7. mention 5961 state machine option
  8. mention 6161 (reducing TIME-WAIT)
  9. incorporate 6429 (ZWP/persist)
  10. look at Tony Sabatini suggestion for describing DO field
  11. clearly specify treatment of reserved bits (see TCPM thread on EDO draft April 25, 2014)
  12. look at possible mention of draft-minshall-nagle (e.g. as in Linux)
  13. make sure that clarifications in RFC 1011 are captured
  14. per TCPM discussion, discussion of checking reserved bits may need to be altered from 793
  15. MSL acronym is defined multiple times
5. IANA Considerations

This memo includes no request to IANA. Existing IANA registries for TCP parameters are sufficient.

TODO: check whether entries pointing to 793 and other documents obsoleted by this one should be updated to point to this one instead.

## 6. Security and Privacy Considerations

TODO

See RFC 6093 [7] for discussion of security considerations related to the urgent pointer field.

Editor's Note: Scott Brim mentioned that this should include a PERPASS/privacy review.

## 7. Acknowledgements

This document is largely a revision of RFC 793, which Jon Postel was the editor of. Due to his excellent work, it was able to last for three decades before we felt the need to revise it.

Andre Oppermann was a contributor and helped to edit the first revision of this document.

We are thankful for the assistance of the IETF TCPM working group chairs:

Michael Scharf  
Yoshifumi Nishida  
Pasi Sarolahti

On the TCPM mailing list, and at the IETF 88 meeting in Vancouver, helpful comments, critiques, and reviews were received from (listed alphabetically): David Borman, Yuchung Cheng, Martin Duke, Kevin Lahey, Kevin Mason, Matt Mathis, Hagen Paul Pfeifer, Anthony Sabatini, Joe Touch, Reji Varghese, Lloyd Wood, and Alex Zimmermann.

This document includes content from errata that were reported by (listed chronologically): Yin Shuming, Bob Braden, Morris M. Keesan, Pei-chun Cheng, Constantin Hagemeyer, Vishwas Manral, Mykyta Yevstifeyev, EungJun Yi, Botong Huang.

## 8. References

### 8.1. Normative References

- [1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

### 8.2. Informative References

- [2] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- [3] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.
- [4] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.
- [5] Lahey, K., "TCP Problems with Path MTU Discovery", RFC 2923, September 2000.

- [6] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, March 2007.
- [7] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, January 2011.
- [8] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, February 2012.
- [9] Borman, D., "TCP Options and Maximum Segment Size (MSS)", RFC 6691, July 2012.
- [10] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, February 2015.

Appendix A. TCP Requirement Summary

This section is adapted from RFC 1122.

TODO: this needs to be seriously redone, to use 793bis section numbers instead of 1122 ones, and all 1122 requirements need to be reflected in 793bis text.

RFC EDITOR'S NOTE: 793bis in the heading below should be replaced by the number of this RFC

FEATURE	RFC1122 SECTION	T	D	Y	O	O	T	S	H	O	M	O					
-----	-----	-	-	-	-	-	-	U	U	M	S	L	A	N	N	T	e
Push flag																	
Aggregate or queue un-pushed data	4.2.2.2								x								
Sender collapse successive PSH flags	4.2.2.2							x									
SEND call can specify PUSH	4.2.2.2									x							
If cannot: sender buffer indefinitely	4.2.2.2														x		
If cannot: PSH last segment	4.2.2.2	x															
Notify receiving ALP of PSH	4.2.2.2									x							1
Send max size segment when possible	4.2.2.2		x														

Window					
Treat as unsigned number	4.2.2.3	x			
Handle as 32-bit number	4.2.2.3		x		
Shrink window from right	4.2.2.16			x	
Robust against shrinking window	4.2.2.16	x			
Receiver's window closed indefinitely	4.2.2.17			x	
Sender probe zero window	4.2.2.17	x			
First probe after RTO	4.2.2.17		x		
Exponential backoff	4.2.2.17		x		
Allow window stay zero indefinitely	4.2.2.17	x			
Sender timeout OK conn with zero wind	4.2.2.17				x
Urgent Data					
Pointer indicates first non-urgent octet	4.2.2.4	x			
Arbitrary length urgent data sequence	4.2.2.4	x			
Inform ALP asynchronously of urgent data	4.2.2.4	x			1
ALP can learn if/how much urgent data Q'd	4.2.2.4	x			1
TCP Options					
Receive TCP option in any segment	4.2.2.5	x			
Ignore unsupported options	4.2.2.5	x			
Cope with illegal option length	4.2.2.5	x			
Implement sending & receiving MSS option	4.2.2.6	x			
Send MSS option unless 536	4.2.2.6		x		
Send MSS option always	4.2.2.6			x	
Send-MSS default is 536	4.2.2.6	x			
Calculate effective send seg size	4.2.2.6	x			
TCP Checksums					
Sender compute checksum	4.2.2.7	x			
Receiver check checksum	4.2.2.7	x			
ISN Selection					
Include a clock-driven ISN generator component	4.2.2.9	x			
Secure ISN generator with a PRF component	N/A		x		
Opening Connections					
Support simultaneous open attempts	4.2.2.10	x			
SYN-RCVD remembers last state	4.2.2.11	x			
Passive Open call interfere with others	4.2.2.18				x
Function: simultan. LISTENS for same port	4.2.2.18	x			
Ask IP for src address for SYN if necc.	4.2.3.7	x			
Otherwise, use local addr of conn.	4.2.3.7	x			
OPEN to broadcast/multicast IP Address	4.2.3.14				x
Silently discard seg to bcast/mcast addr	4.2.3.14	x			
Closing Connections					

RST can contain data	4.2.2.12	x			
Inform application of aborted conn	4.2.2.13	x			
Half-duplex close connections	4.2.2.13		x		
Send RST to indicate data lost	4.2.2.13	x			
In TIME-WAIT state for 2xMSL seconds	4.2.2.13	x			
Accept SYN from TIME-WAIT state	4.2.2.13		x		
Retransmissions					
Jacobson Slow Start algorithm	4.2.2.15	x			
Jacobson Congestion-Avoidance algorithm	4.2.2.15	x			
Retransmit with same IP ident	4.2.2.15		x		
Karn's algorithm	4.2.3.1	x			
Jacobson's RTO estimation alg.	4.2.3.1	x			
Exponential backoff	4.2.3.1	x			
SYN RTO calc same as data	4.2.3.1		x		
Recommended initial values and bounds	4.2.3.1		x		
Generating ACK's:					
Queue out-of-order segments	4.2.2.20		x		
Process all Q'd before send ACK	4.2.2.20	x			
Send ACK for out-of-order segment	4.2.2.21			x	
Delayed ACK's	4.2.3.2		x		
Delay < 0.5 seconds	4.2.3.2	x			
Every 2nd full-sized segment ACK'd	4.2.3.2	x			
Receiver SWS-Avoidance Algorithm	4.2.3.3	x			
Sending data					
Configurable TTL	4.2.2.19	x			
Sender SWS-Avoidance Algorithm	4.2.3.4	x			
Nagle algorithm	4.2.3.4		x		
Application can disable Nagle algorithm	4.2.3.4	x			
Connection Failures:					
Negative advice to IP on R1 retxs	4.2.3.5	x			
Close connection on R2 retxs	4.2.3.5	x			
ALP can set R2	4.2.3.5	x			1
Inform ALP of R1<=retxs<R2	4.2.3.5		x		1
Recommended values for R1, R2	4.2.3.5		x		
Same mechanism for SYNs	4.2.3.5	x			
R2 at least 3 minutes for SYN	4.2.3.5	x			
Send Keep-alive Packets:					
- Application can request	4.2.3.6			x	
- Default is "off"	4.2.3.6	x			
- Only send if idle for interval	4.2.3.6	x			
- Interval configurable	4.2.3.6	x			
- Default at least 2 hrs.	4.2.3.6	x			
- Tolerant of lost ACK's	4.2.3.6	x			

IP Options					
Ignore options TCP doesn't understand	4.2.3.8	x			
Time Stamp support	4.2.3.8			x	
Record Route support	4.2.3.8			x	
Source Route:					
ALP can specify	4.2.3.8	x			1
Overrides src rt in datagram	4.2.3.8	x			
Build return route from src rt	4.2.3.8	x			
Later src route overrides	4.2.3.8		x		
Receiving ICMP Messages from IP					
Dest. Unreach (0,1,5) => inform ALP	4.2.3.9	x			
Dest. Unreach (0,1,5) => abort conn	4.2.3.9		x		x
Dest. Unreach (2-4) => abort conn	4.2.3.9		x		
Source Quench => slow start	4.2.3.9		x		
Time Exceeded => tell ALP, don't abort	4.2.3.9		x		
Param Problem => tell ALP, don't abort	4.2.3.9		x		
Address Validation					
Reject OPEN call to invalid IP address	4.2.3.10	x			
Reject SYN from invalid IP address	4.2.3.10	x			
Silently discard SYN to bcast/mcast addr	4.2.3.10	x			
TCP/ALP Interface Services					
Error Report mechanism	4.2.4.1	x			
ALP can disable Error Report Routine	4.2.4.1		x		
ALP can specify TOS for sending	4.2.4.2	x			
Passed unchanged to IP	4.2.4.2		x		
ALP can change TOS during connection	4.2.4.2		x		
Pass received TOS up to ALP	4.2.4.2			x	
FLUSH call	4.2.4.3			x	
Optional local IP addr parm. in OPEN	4.2.4.4	x			
-----					

FOOTNOTES: (1) "ALP" means Application-Layer program.

Author's Address

Wesley M. Eddy (editor)  
 MTI Systems  
 US

Email: wes@mti-systems.com